PL/MMIX

A structured assembly language for Knuth's MMIX computer,
inspired by Wirth's PL/360.

Filip Stappers

May 14, 2026

Contents

1 Introduction **5**
1.1 Relation to PL/360 . . . 5
1.2 Other PL/MMIX designs . . . 5
1.3 Source files . . . 5
1.4 Available Software . . . 5
1.5 About this manual . . . 6

2 Lexical Conventions **7**
2.1 Comments . . . 7
2.2 Identifiers and Labels . . . 7
2.3 Registers . . . 7
2.4 Immediate constants . . . 7
2.4.1 Character literals . . . 8
2.4.2 String literals and UTF-8 . . . 8
2.5 Reserved Words . . . 8

3 Program Structure **10**
3.1 Implicit main body . . . 10
3.2 Two-segment memory layout . . . 10
3.3 Declarations and statements at the top level . . . 10

4 Register Declarations **12**

5 Procedures **14**
5.1 Calling procedures and returning values . . . 15
5.1.1 Value-expression returns . . . 15
5.1.2 Inline string literals as call arguments . . . 16
5.2 Local scope block . . . 16

6 Data Layout Directives **18**
6.1 GREG – global register allocation . . . 18
6.2 BYTE/WYDE/TETRA/OCTA – emit data words . . . 19
6.3 SEGMENT – switch to a named segment . . . 19
6.4 POOL – emit a literal pool . . . 20
6.5 LOC – set location counter* . . . 20
6.6 RESERVE – reserve uninitialised space . . . 20

7 Namespace and Composition Directives **22**
7.1 PREFIX – assembler namespace . . . 22
7.2 LOCAL – assert a local register . . . 22
7.3 INCLUDE – source composition . . . 22

8 Statements **24**
8.1 Assignment . . . 24
8.1.1 Binary operators . . . 24
8.1.2 Unary operators . . . 25
8.1.3 Rounding mode . . . 26
8.2 Conditional and Zero-Set Statements (CS/ZS family) . . . 26
8.3 Memory Access . . . 26
8.3.1 Label-based addressing and base-register selection . . . 28
8.4 Address-of Operator . . . 28

8.5 Wyde-Immediate Bit Updates . . . 29

9 Special Registers and Predefined Constants 30
9.1 GET and PUT . . . 30
9.2 Register inventory . . . 30
9.3 Predefined constants . . . 31

10 Control Flow 33
10.1 If statement . . . 33
10.2 Postfix if . . . 33
10.3 While statement . . . 34
10.4 Loop statement . . . 34
10.5 Do... while statement . . . 34
10.6 Break and continue . . . 35
10.7 Branch hints . . . 35
10.8 Conditions . . . 35
10.9 Goto statement . . . 37

11 Cache and Memory-Order Statements* 38
11.1 Cache prefetch and writeback . . . 38
11.2 Memory fence . . . 38

12 Compare-and-Swap* 39

13 Register-Stack Save and Restore* 40

14 Trips, Traps, and Resume* 41
14.1 Trap (OS service request) . . . 41
14.2 Trip (user exception) . . . 41
14.3 Resume . . . 42

15 Translation Lookaside Buffer Management (LDVTS)* 43

16 Literate Programming with `.plw` Files 44
16.1 Tools . . . 44
16.2 File format . . . 44
16.3 Named chunks . . . 44
16.4 Comment conventions . . . 44
16.5 #line directives . . . 44
16.6 Weave output . . . 45
16.7 Cross-reference index . . . 45
16.8 Complete example: TPK algorithm in `.plw` . . . 45

A Grammar Summary 58

B TPK Algorithm: Woven Output 61
1. The TPK Algorithm in PL/MMIX. . . . 61
2. Program structure. . . . 61
3. Reading the inputs. . . . 62
4. Computing and printing in reverse. . . . 62
5. The floating-point input parser. . . . 63
6. The output formatter. . . . 66
7. Build and weave. . . . 71

C Bootstrap Tangle: Woven Output **72**
1. Introduction. . . . 72
2. Error handling. . . . 73
3. Constants and limits. . . . 73
4. Data segment. . . . 74
5. Main program. . . . 75
6. Procedure inventory and tangled layout. . . . 76
7. Reading the input. . . . 76
8. String utilities. . . . 78
9. File and include operations. . . . 80
10. Chunk table operations. . . . 87
11. Parsing the literate web source. . . . 88
12. Expanding chunks. . . . 94
13. Output buffer and \#line directives. . . . 98

D Bootstrap Weave: Woven Output **102**
1. 1. Introduction. . . . 102
2. Constants and limits. . . . 103
3. Data segment. . . . 105
4. Main program. . . . 108
5. Procedure inventory and tangled layout. . . . 109
6. Reading the input. . . . 109
7. Character predicates. . . . 112
8. String utilities. . . . 113
9. Section table helpers. . . . 118
10. Parsing the literate source. . . . 119
11. Hash tables for unique-name lookup. . . . 124
12. Unique chunk, label, constant, and register tables. . . . 127
13. Sorting cross-reference tables. . . . 146
14. Output helpers. . . . 148
15. Anchor emitters. . . . 155
16. Rendering prose. . . . 157
17. Rendering code bodies. . . . 163
18. Emitting the document. . . . 175
19. Cross-reference index. . . . 180
20. Error handling. . . . 185
21. Showing the work. . . . 186

E Bootstrap Compiler: Woven Output **187**
1. Introduction. . . . 187
2. Constants and limits. . . . 188
3. Data segment. . . . 192
4. Main program. . . . 193
5. Reading the source. . . . 194
6. String utilities. . . . 195
7. Integer conversion. . . . 197
8. Output helpers. . . . 199
9. Error handling. . . . 200
10. Token kinds. . . . 201
11. Lexer: overview and main loop. . . . 207
12. Deferred scanners: identifiers and keywords. . . . 216
13. Numeric literals and scaled-add tokens. . . . 218

14. String and character literals. 221
15. Dollar-prefixed terminals. 224
16. Dot-prefixed size suffixes. 226
17. Token emission and helpers. 226
18. Parser: cursor and node allocation. 240
19. Parser: program and simple top-level declarations. 249
20. Parser: constant expressions and data declarations. 253
21. Parser: procedure declarations. 259
22. Generator: design overview. 293
23. Generator: top-level skeleton. 294
24. Driver banner. 317
25. Generator: procedure bodies. 339
26. Generator: control flow. 377

F MMIX and PL/MMIX Side-by-Side **393**

1 Introduction

"The tools we use have a profound and devious influence on our thinking habits, and therefore on our thinking abilities."

— Edsger W. Dijkstra, *How Do We Tell Truths That Might Hurt?* (1975)

PL/MMIX is a structured assembly language for Knuth's 64-bit RISC architecture MMIX [2, 3], conceived in the same spirit as Wirth's PL/360 [5]: expose the hardware directly, add enough syntactic sugar to write readable programs, and keep the mapping from source to machine instructions transparent.

PL/MMIX was designed by the author, using various LLMs (Claude Sonnet 4.6, Claude Opus 4.6 and 4.7, GPT 5.4 and 5.5).

Design principles.

- PL/MMIX is a thin structured layer over MMIX: each construct expands to one instruction or a short, fixed instruction sequence.
- Registers, data sizes, and memory-placement directives remain explicit in the source, so generated MMIXAL stays easy to inspect.
- MMIXAL concepts such as labels, `greg`, `loc`, and special registers are reused directly instead of being hidden behind a second model.
- Facilities for literate programming [4] are available.

1.1 Relation to PL/360

Wirth's PL/360 [5] introduced structured control flow and a readable assignment notation for the IBM System/360 assembler in 1968. PL/MMIX does the same for MMIX, updated for a 64-bit load/store architecture with IEEE 754 floating point, a windowed register file, and explicit memory-ordering primitives.

1.2 Other PL/MMIX designs

While working on this project, the author learned that W. de Boer also designed and implemented a PL/MMIX language in 2025 [1] . His language is closer to the C language, supplies a few more programmer comfort features and does not support literate programming.

1.3 Source files

Plain PL/MMIX source files use the extension `.plm`. Literate PL/MMIX source files use the extension `.plw` (see Section 16). The compiler (`plmmix-compiler`) emits MMIX assembly (`.mms`), which is then assembled with `mmixal` into an `.mmo` binary and run with the `mmix` simulator.

1.4 Available Software

The PL/MMIX tools are distributed as `plmmix.zip`, containing:

- `compiler/` — The PL/MMIX compiler (`plmmix-compiler.plw`), tangle (`plmmix-tangle.plw`), and weave (`plmmix-weave.plw`), all written in PL/MMIX.
- `examples/` — Example programs: `gcd.plw` (Euclidean algorithm), `tpk.plw` (Knuth & Pardo's TPK algorithm), plus PL/MMIX reimplementations of the MMIX Masters problems (`mmixmasters_plmmix/`) and programs from Knuth's MMIX fascicle (`mmix_fasc_plmmix/`), all as `.plm` source files.

Compiling and running a program

A literate PL/MMIX source (`.plw`) is processed in three steps: tangle, compile, and assemble. All PL/MMIX tools follow the same I/O convention: the source file content is supplied on standard input (via the MMIX simulator's `-f` flag), the source file path is passed as the first command-line argument, and the output is written to standard output.

```
# Step 1 - tangle: extract plain source from a literate .plw file
mmix -fmy_prog.plw  plmmix-tangle.o    my_prog.plw  > my_prog.plm

# Step 2 - compile: emit MMIXAL assembly
mmix -fmy_prog.plm  plmmix-compiler.o  my_prog.plm  > my_prog.mms

# Step 3 - assemble and run
mmixal -o my_prog.mmo  my_prog.mms
mmix my_prog.mmo
```

For non-literate (`.plm`) sources, step 1 is omitted. The weave tool follows the same pattern and produces a TEX file on standard output.

1.5 About this manual

Sections marked with an asterisk (*) cover advanced topics that are not required for basic programming with PL/MMIX. These include low-level memory ordering, concurrent programming primitives, and exception handling. Readers new to PL/MMIX may safely skip these sections on a first reading.

The appendices also include complete woven outputs for the literate TPK example and for the three bootstrapped tools: `plmmix-tangle.plw`, `plmmix-weave.plw`, and `plmmix-compiler.plw`. Finally, there is a side-by-side comparison of the original MMIXmaster programs and their PL/MMIX implementation.

2 Lexical Conventions

"The limits of my language mean the limits of my world."
— Ludwig Wittgenstein, *Tractatus Logico-Philosophicus* (1921)

2.1 Comments

PL/MMIX has one comment form: a line comment introduced by `#`. It is stripped before assembly and never appears in the MMIX output.

```
LineComment ::= "#" {any character} newline
```

The comment extends from the `#` to the end of the line and is valid anywhere whitespace is permitted. The `#` character followed immediately by a hex digit is *not* a comment—it introduces a hex immediate constant such as `#FF` or `#2000000000000000`.

2.2 Identifiers and Labels

```
Label ::= IdentStart { IdentCont }
IdentStart ::= Letter | NonAsciiByte
IdentCont ::= IdentStart | DecDigit
Letter ::= "A" | ...| "Z" | "a" | ...| "z" | "_"
NonAsciiByte ::= any source byte with the high bit set (0x80-0xFF)
```

Labels are case-sensitive. A *label declaration* is a label followed by a colon (`L:`); it introduces an assembler symbol at the current location.

Non-ASCII identifier bytes. The bootstrap compiler recognises identifiers bytewise rather than by Unicode character classes. In addition to ASCII letters and underscores, any byte with the high bit set may appear in an identifier, so UTF-8 source text passes through the lexer without a Unicode table. The compiler preserves those bytes when emitting MMIXAL names; it does not normalise or mangle them. For maximum portability with downstream MMIX tools, ASCII identifiers remain the safest choice.

2.3 Registers

```
Register ::= "$" DecDigit { DecDigit }   (0-255)
```

MMIX has 256 general-purpose registers, `$0`–`$255`. Within a procedure body, `$0`–`$n-1` are the parameters; `$n` and above are temporaries.

2.4 Immediate constants

```
SmallImm ::= DecDigit { DecDigit }   (0-255)
| "#" HexDigit HexDigit   (#00-#ff)
WydeImm ::= DecDigit { DecDigit }   (0-65535)
| "#" HexDigit^4   (#0000-#ffff)
Immediate ::= [ "-" ] DecDigit { DecDigit }   (any value, optionally negated)
| "#" HexDigit { HexDigit }   (up to 16 hex digits, unsigned)
| CharLit   (character ordinal)
CharLit ::= "'" ( Char | EscapeSeq ) "'"
```

Note

A leading `"-"` is allowed before a decimal *Immediate* in assignment expressions, **`register`** constant definitions, **`data`** literals, and **`greg`** initialisers. The value is treated as a 64-bit two's complement integer. For example, `$x = -1;` loads `#FFFFFFFFFFFFFFFF`.
When an *Immediate* is used in an assignment (`$x = Immediate;`), the compiler emits the minimal sequence of `SETH`/`SETMH`/`SETML`/`SETL`/`INC*` instructions needed to load the full 64-bit constant.

2.4.1 Character literals

A single-quoted *CharLit* stands for the Unicode codepoint of the enclosed character and is accepted wherever an *Immediate* is accepted: assignment expressions, arithmetic immediates, and data initialisers. The following escape sequences are supported:

Escape	Meaning
`'\n' '\t' '\r' '\\' '\'' '\0' ...`	Standard C-style
`'\xNN'`	Hex byte (0x00–0xFF)
`'\uNNNN'`	Unicode BMP codepoint (4 hex digits)
`'\UNNNNNNNN'`	Unicode full range (8 hex digits)

Non-ASCII characters may also appear directly in the source (UTF-8 encoded).

```
$d = $d + '0';        # ADD $d,$d,48  -- remainder -> ASCII digit
$x = '\n';            # load 10 (newline)
$buf[0].b = '\t';     # STB 9,$buf,0  -- store tab character
byte NL '\n';         # BYTE 10
$x = '\u03B1';        # load 945 (U+03B1, GREEK SMALL LETTER ALPHA)
$x = 'α';             # same, using direct UTF-8 in source
```

2.4.2 String literals and UTF-8

A double-quoted string in a **`byte`** declaration is emitted as a sequence of UTF-8 bytes. Non-ASCII characters expand to their multi-byte UTF-8 encodings, so the resulting data is valid UTF-8 text. Unicode escape sequences `\uNNNN` and `\UNNNNNNNN` are also accepted inside string literals. No implicit zero terminator is added: if a C-style NUL-terminated string is required, write the trailing zero explicitly as another data literal.

```
byte Greeting "Café\n", 0;
byte Pi       "π = 3.14\n", 0;
```

In **`wyde`**/**`tetra`**/**`octa`** declarations, each character expands to its Unicode codepoint value as a single integer (rarely useful for non-ASCII text).

2.5 Reserved Words

The following identifiers are recognised as keywords by the lexer and may not be used as ordinary labels or register aliases:

Procedures and blocks

`def end begin return call`

Control flow

`if elif else while do loop break continue goto`

Branch hints

`likely unlikely`

Polarity tests
`is neg zero pos odd nonneg nonzero nonpos even`

Declarations
`register const greg local include prefix`

Data and segments
`data code segment pool loc reserve byte wyde tetra octa`

Floating-point and rounding
`rounding nearest up down`

Register-stack and synchronisation
`save unsave preld prego prest syncd syncid sync cswap ldvts`

Traps and resume
`trap trip resume`

Named binary operators
`andn orn nand nor nxor mor mxor mux sadd bdif wdif tdif odif cmp cmpu`

The following lexemes are reserved at the operator level and must not appear as identifier characters: `?=`, `!=`, `==`, `<=`, `>=`, `<u`, `<=u`, `>u`, `>=u`, `==.`, `!=.`, `<.`, `<=.`, `>.`, `>=.`, plus the wyde-update suffixes `orh=`, `orhh=`, `orhhh=`, `orl=`, `andnh=`, `andnhh=`, `andnhhh=`, `andnl=`.

Special-register letters. The single letters `B`, `D`, `E`, `H`, `J`, `M`, `R`, `C`, `N`, `O`, `S`, `I`, `T`, `K`, `Q`, `U`, `V`, `G`, `L`, `A`, `F`, `P`, `W`, `X`, `Y`, `Z` and the digraphs `BB`, `TT`, `WW`, `XX`, `YY`, `ZZ` are reserved *only* when they immediately follow `$r` (e.g. `$rA`, `$rBB`): they name the MMIX special registers in those positions and are unavailable as ordinary identifiers there. Elsewhere they are normal labels.

3 Program Structure

"The structure of a program should reflect the structure of the problem it solves."
— Niklaus Wirth, *Algorithms + Data Structures = Programs* (1976)

```
Program ::= { ProcDecl | DataDecl | SegmentDirective | LocDirective |
ReserveDirective
              | RegisterDecl | ConstDecl | PrefixDirective | LocalDecl |
IncludeDirective | PoolDirective
              | Statement }
```

A PL/MMIX program is a flat sequence of declarations and statements at the top level. Procedure declarations introduce named callable units. Statements outside any procedure form the implicit *main body*.

3.1 Implicit main body

Statements that appear at the top level, outside any `def` block, are collected in order into the *implicit main body*—the program's entry point. There is no explicit `main` keyword; execution begins at the first instruction of the assembled code segment. The main body typically ends with `trap 0, HALT, 0;` to terminate the process.

```
# Top-level statements form the implicit main body.
register $n = $0;
$n = 42;
trap 0, HALT, 0;    # terminate
```

Procedure declarations (`def` … `end`) may appear anywhere in the source interleaved with top-level statements; the compiler emits them before the main body in the assembled output so that `PUSHJ` can reach them with a relative branch.

3.2 Two-segment memory layout

MMIX's 64-bit address space is conventionally split into a code segment and a data segment:

Segment	Base address	Contents
Code	`#100`	executable instructions
Data	`#2000000000000000`	global variables, strings, tables

The standard idiom uses `segment` to switch between them:

```
segment data;           # switch to data segment
greg $DataBase;         # GREG @ -- base register for data
byte greeting "Hello\n";

segment code;           # switch back to code segment
greg $CodeBase;         # GREG @ -- base register for code
# ... instructions follow
```

The compiler automatically relocates any `def` blocks that appear inside the data segment to the code segment (immediately after the `segment code;` line), so the source can be organised in any order. See Section 6 for details.

3.3 Declarations and statements at the top level

Any mix of the following may appear at the top level:

- **Register declarations** (`register`) — introduce register or constant names visible in all subsequent code, including procedure bodies. See Section 4.
- **Procedure declarations** (`def` … `end`) — define named callable units; emitted before the main body. See Section 5.
- **Data-layout directives** (`greg`, `byte`, `wyde`, `tetra`, `octa`, `segment`, `loc`, `reserve`) — allocate global registers and static storage. See Section 6.
- **Namespace and composition directives** (`prefix`, `local`, `include`) — control assembler symbol scoping and source composition. See Section 7.
- **Statements** — all executable statement forms; collected into the implicit main body in source order. See Sections 8 and 10.

The ordering rules are simple: a `register` declaration must appear before the code that uses it; all other declarations and statements may be freely intermixed.

4 Register Declarations

"The beginning of wisdom is to call things by their proper name."
— Confucius

```
RegisterDecl ::= register RegName "=" Register ";"   (IS - name a register,
explicit)
| register RegName ";"   (IS - name next free register, auto)
| register RegName { "," RegName } ";"   (IS - multi-name auto-assign)

ConstDecl ::= const Label "=" ConstExpr ";"   (IS - define a compile-time constant)

RegName ::= "$" IdentStart { IdentCont }   ($identifier, e.g. $n, $ptr - required
for register aliases)

ConstExpr ::= ConstAtom { ("+" | "-" | "*" | "/") ConstAtom }
| "(" ConstExpr ")"

ConstAtom ::= [ "-" ] Immediate | ConstAliasName

RegOperand ::= Register | RegName   (register alias, resolving to a register)
```

Generates an MMIX `IS` assembler directive.

Register aliases (`register $x = $5;`) make `$x` synonymous with `$5` in every position where a register is accepted, including as the *left-hand side* of an assignment. This is *register alias transparency*: the compiler resolves the alias to the underlying register number at parse time, exactly as MMIXAL resolves IS-defined names at assembly time. Throughout this manual, wherever the grammar says *Register*, an alias name that resolves to a register may appear in its place. Register aliases must use a `$`-prefixed *RegName* (e.g. `$n`, `$ptr`).

Compile-time constants (`const`) make `N` a symbolic constant usable wherever an *Immediate* or *SmallImm* is accepted, and also as a pure right-hand-side assignment expression. The right-hand side may be a compile-time *ConstExpr*: an arithmetic expression over integer literals and previously-defined constant aliases, using `+`, `-`, `*`, and `/` (integer division, truncating toward zero). Parentheses are supported. The expression is fully evaluated at compile time and emitted as a single `IS` constant. Constant names are plain identifiers without a `$` prefix, which visually distinguishes them from register aliases.

Aliases are file-scoped in terms of name resolution: an alias defined before a procedure is visible inside that procedure's body. However, the compiler only emits the `IS` assembler directive for aliases declared at the top level. Aliases declared inside a procedure body (including named parameters) are used purely as compile-time substitutions and produce no assembler output, which avoids "symbol already defined" errors when the same alias name is reused in multiple procedures. The only positions that require literal `$N` syntax are unnamed *ProcParam*s and the `local` directive.

Auto-register assignment

The bare form `register $x;` (without `= $N`) assigns the next free register automatically. The compiler maintains a counter:

- At the top level, the counter starts at 0.
- On entry to a procedure body, the counter resets to *len(params)* (the number of parameters), then restores on exit.
- An explicit `register $x = $N;` advances the counter past `$N` (it only moves forward, never back).

This lets sequential locals be written concisely while still allowing deliberate jumps when needed:

```
def Add($U, $V, $W, $n):    # auto: $U=$0, $V=$1, $W=$2, $n=$3; counter
    -> 4
  register $j;              # $4
  register $k;              # $5
  register $wj;             # $6
  register $tmp;            # $7
  ...
end
```

When a register must be chosen explicitly (reuse of a parameter, `$255`, or a deliberate gap), the explicit form is still available and does not conflict with the auto form.

```
register $t = $2;

# Compile-time constants with arithmetic
const BufSize    = 4096;
const HdrSize    = 16;
const DataOffset = BufSize + HdrSize;   # IS 4112
const Mask       = (BufSize - 1);       # IS 4095

# Named parameters introduce register aliases automatically
def max2($left, $right):
  if $left < $right:
    $t = $left;
    $left = $right;
    $right = $t;
  end
  return $left;
end

# Register alias as address base
register $base = $20;
$5 = [$base + $3].o;    # LDO $5,$20,$3
$6 = $base[$3].o;       # same load, shorthand form
```

5 Procedures

"Le second, de diviser chacune des difficultés que j'examinerais, en autant de parcelles qu'il se pourroit, et qu'il seroit requis pour les mieux résoudre."
— René Descartes, *Discours de la méthode* (1637)

```
ProcDecl ::= def Label "(" ProcParams ")" [ "->" ProcParams ] ":"
             StatementList
             end
ProcParams ::= [ ProcParam { "," ProcParam } ]
ProcParam ::= Register | ParamName "=" Register | ParamName
ParamName ::= Label | RegName
```

Parameters must be `$0`, `$1`, ..., `$n-1` in order. An empty parameter list `()` is valid. Registers above `$n-1` are fresh temporaries local to the call frame.

Each parameter may be given a name. The `name=$N` form is explicit; the bare `name` form auto-assigns the register by position (`$0` for the first parameter, `$1` for the second, etc.). Since parameters are always in order, the register is redundant—bare names are the preferred style. A named parameter acts exactly like `register $name = $N;` at the top of the procedure body. Named, unnamed (`$N`), and auto-named parameters may be mixed freely.

Output parameters. The optional `->` clause names the output (return-value) registers. Output parameters are assigned register numbers immediately after the input parameters; the `return` statement copies them to `$0`, `$1`, ... before the `POP`. Output parameter names are available as register aliases throughout the procedure body.

```
def absdiff($x, $y):
  register $d;
  if $x >= $y:
    $d = $x - $y;
  else:
    $d = $y - $x;
  end
  return $d;
end

def divmod($a, $b) -> $q, $r:
  $q = $a / $b;
  $r = $rR;
  return $q, $r;
end
```

A *StatementList* is a sequence of label declarations and statements:

```
StatementList ::= { LabelDecl | Statement }
LabelDecl ::= Label ":"
```

Labels may appear freely inside procedure bodies and at the top level. If the same label name is used in two or more different procedure bodies, the compiler automatically mangles each occurrence to `ProcName__Label`, so that cross-procedure `goto` statements targeting a uniquely-named label continue to work without modification.

5.1 Calling procedures and returning values

```
CallStatement ::= RegList "=" call Label "(" ArgList ")" ";"   (PUSHJ, n returns)
| call Label "(" ArgList ")" ";"   (PUSHJ, void)
| RegList "=" call "[" RegOperand "]" "(" ArgList ")" ";"   (PUSHGO, n returns)
| call "[" RegOperand "]" "(" ArgList ")" ";"   (PUSHGO, void)

ReturnStatement ::= return ReturnExpr { "," ReturnExpr } ";"   (POP n,0)
| return ";"   (POP 0,0)

ReturnExpr ::= RegOperand   (used directly as POP value)
               | [ "-" ] Immediate
               | CharLit
               | Label   (const alias)
               | call Label "(" ArgList ")"   (call, result assigned to return reg)

RegList ::= RegOperand { "," RegOperand }
ArgList ::= [ ArgItem { "," ArgItem } ]
ArgItem ::= any single assignment atom accepted by the bootstrap compiler, or a
string literal (see below)
```

MMIX's PUSHJ/PUSHGO save the register window and jump; the compiler inserts POP n,0 at return. Multiple return values are placed in $0..$n-1 in the callee and received into the *RegList* in the caller. A direct label address may also be passed as \&Label; it is lowered with the same GETA or auto-GREG strategy used by address-of assignment.

```
register $m, $n, $result;
# Single return value
$result = call max2($m, $n);     # PUSHJ + POP 1,0

register $a, $b, $q, $r;
# Multiple return values
$q, $r = call divmod($a, $b);    # PUSHJ + POP 2,0

register $msg;
# Void call
call print($msg);              # PUSHJ + POP 0,0

# Pass a direct label address
call print(&Banner);           # GETA/GREG into argument register, then
    PUSHJ

register $arg, $ret, $fn;
# Indirect call via function pointer in $fn
$ret = call [$fn]($arg);         # PUSHGO
```

5.1.1 Value-expression returns

When all *ReturnExpr*s are registers or register aliases, **return** behaves exactly as before: those registers become the POP values and no -> clause is required.

When any *ReturnExpr* is a non-register atom—an immediate constant, character literal, compile-time constant alias, or a **call** expression—the function must declare its return registers with a -> clause. Each expression is then assigned *positionally* to the corresponding declared return register, after which a standard POP n,0 is emitted. The expression count must equal the number of names in the -> clause.

A **call** expression in return position emits the full call sequence (argument setup, PUSHJ/PUSHGO, rJ restore); the callee's single return value lands in the call-base register $19, which is then

moved to the declared return register if it differs. Only single-value call results are supported in this position.

```
def IsDigit($ch=$0) -> $r:
  register $tmp;
  $tmp = $ch - '0';
  if $tmp is neg:   return 0;   end
  $tmp = $ch - '9';
  if $tmp is nonpos: return 1;  end
  return 0;
end
```

Each `return 0;` and `return 1;` compiles to `$r = 0; POP 1,0` and `$r = 1; POP 1,0` respectively, eliminating the label-and-goto chains that the purely register-based form would require. Mixing register and non-register expressions in the same **return** is allowed: `return 0, $rR;` assigns 0 to the first declared return register and uses `$rR` directly as the second POP value.

5.1.2 Inline string literals as call arguments

For call arguments, the compiler accepts the same single-atom forms that appear on the right-hand side of an assignment: registers, special registers, immediates, floating literals, labels, `&label`, load forms, unary operators. In addition, a string literal is accepted directly at any *ArgItem* position; it is lowered as if the programmer had written a hidden **byte** declaration in the data segment and passed its address with `&`.

A bare string literal as a call argument synthesises a fresh data-segment declaration of the form

```
byte __InlineStr<N> "...", 0;
```

where `<N>` is a per-program counter. The argument is then passed as `\&__InlineStr<N>`, i.e. via GETA or an auto-GREG just like any other address-of operand. The synthesised declarations are spliced into the data segment immediately before the first `segment code;` directive (or before the first **def**, whichever comes first), so they are reachable by GETA from the code segment without manual layout. This is purely a parser-side desugaring; once lowered, an inline string is indistinguishable from a hand-written one and incurs no extra runtime cost. Repeated identical literals are not coalesced; each occurrence allocates a fresh label.

```
register $out = $255;
def Print($p):
  $out = $p;
  trap 0, FPUTS, STDOUT;
  return;
end
def Main():
  call Print("Hello, world!\n");    # synthesises __InlineStr0
  call Print("second line\n");      # synthesises __InlineStr1
  trap 0, HALT, 0;
end
```

5.2 Local scope block

```
BeginBlock ::= begin ":" StatementList end
```

A **begin**. . . **end** block introduces a nested local scope. Register aliases declared inside the block are scoped to the block and do not escape into the enclosing body. There is no runtime overhead: the generated code is identical to placing the statements directly in the enclosing body.

```
def foo($a, $b) -> $result:
  $result = $a + $b;
  begin:
    register $tmp;          # $tmp exists only in this block
    $tmp = $result * $result;
    $result = $result + $tmp;
  end
  return $result;
end
```

6 Data Layout Directives

"Smart data structures and dumb code works a lot better than the other way around."
— Eric S. Raymond, *The Cathedral and the Bazaar* (1999)

These declarations generate MMIX assembler directives, not runtime instructions.

6.1 GREG – global register allocation

```
GregDecl ::= greg GregName ( "[" ConstExpr "]" | [ "=" GregInit ] ) ";"
GregInit ::= Label [ ("+" | "-") ConstExpr ]
| ConstExpr
| [ "-" ] FloatLit
FloatLit ::= Digit+ "." Digit+ [ ("e" | "E") [ "+" | "-" ] Digit+ ]
ConstExpr ::= ConstTerm { ("+" | "-") ConstTerm }
ConstTerm ::= ConstAtom { ("*" | "/") ConstAtom }
ConstAtom ::= [ "-" ] Immediate | "(" ConstExpr ")"
GregName ::= Label | RegName
```

Generates a `GREG` directive, allocating one global register initialised to the given value. Without `= GregInit`, the initial value is `@` (the current location counter). Both plain labels and `$`-prefixed register-style names are accepted; the `$name` style is preferred, so global registers read like registers at every use site. The leading `$` is stripped when MMIXAL is emitted, producing an ordinary assembler label such as `DataBase GREG @`.

A *GregInit* may be a pure numeric constant expression, folded to a single integer at compile time; a label optionally offset by a constant expression (passed through to MMIXAL verbatim); or a *FloatLit* floating-point literal, folded to its IEEE 754 double-precision bit pattern at compile time.

```
greg $DataBase;               # GREG @ -- base for current location
greg $CodeBase = #100;        # GREG #100
greg $j = 24*8;               # GREG #C0   (numeric constant expression
    )
greg $Xn = Xbuf + 54*8;      # GREG Xbuf+432  (label-relative)
```

Floating-point constant GREGs. A *FloatLit* initialiser requires at least one digit before and after the decimal point, with an optional decimal exponent:

```
greg $threshold_f = 400.0;    # GREG #4079000000000000  (400.0 IEEE
    754)
greg $five_f      = 5.0;      # GREG #4014000000000000
greg $neg_half    = -0.5;     # GREG #BFE0000000000000
greg $tiny        = 1.5e-10; # GREG #3DDB7CDFD9D7BDBB (approx)
```

The compiler converts the literal to its 64-bit IEEE 754 representation at compile time and emits `GREG #xxxx`. The program loader pre-initialises the global register with that bit pattern, so no runtime `FLOT` instruction is needed. This matches the behaviour of native MMIXAL, where `GREG 400.0` is also assembled directly to the IEEE 754 bit pattern.

Buffer declaration: `greg $Name[N]`. The bracketed form `greg $Name[N]` is a compact shorthand for the common pattern of anchoring a register at a buffer and immediately reserving the buffer's storage:

```
greg $InputBuf[128];    # InputBuf GREG @; LOC @+128  -- 128-byte
    buffer
```

is exactly equivalent to:

```
greg $InputBuf;          # InputBuf GREG @
reserve 128;             #          LOC @+128
```

The `= GregInit` and `[N]` forms are mutually exclusive: `= GregInit` sets the register to an arbitrary compile-time value and does not advance the location counter, while `[N]` fixes the register at `@` and advances it by N bytes.

The greg+data idiom. A common pattern is to anchor a global register at the start of a data buffer and then immediately declare the buffer contents:

```
greg $DataBase;          # GREG @  -- $DataBase holds this address
octa Table 0, 0, 0;      # three octabytes follow
```

The register captures the current location counter regardless of the data label that follows. If you intentionally reuse the same stem (`greg $Table;` followed by `octa Table ...;`), the compiler also suppresses the duplicate assembler label on the emitted `OCTA`/`BYTE`/`TETRA`/`WYDE` line, avoiding the "symbol already defined" error that MMIXAL would otherwise produce.

6.2 BYTE/WYDE/TETRA/OCTA – emit data words

```
DataDecl ::= byte Label DataLiteral { "," DataLiteral } ";"
| wyde Label DataLiteral { "," DataLiteral } ";"
| tetra Label DataLiteral { "," DataLiteral } ";"
| octa Label DataLiteral { "," DataLiteral } ";"

DataLiteral ::= Immediate | Label | StringLit

StringLit ::= " { char | EscapeSeq } "
```

Emits one or more data words using the MMIX assembler directives `BYTE`, `WYDE`, `TETRA`, or `OCTA`. These direct size keywords are the canonical source form.

String literals. A double-quoted string literal may appear as a *DataLiteral*. The compiler expands it to a sequence of byte values at compile time, one per character, recognising standard C-style escape sequences (`\n`, `\t`, `\r`, `\\`, `\"`, `\0`, etc.). String literals are only meaningful with size `b`; with other sizes each character still produces one entry, which is rarely intended. As with the byte-string examples above, the string contributes exactly its decoded bytes; no implicit trailing zero is appended.

```
byte greeting "Hello\n";                  # BYTE 72,101,108,108,111,10
octa Pi       #400921FB54442D18;          # 64-bit IEEE pi
tetra Table   0, 1, 2, 3;                # four tetras
byte cstr     "word", 0;                  # explicit NUL-terminated
   string
```

6.3 SEGMENT – switch to a named segment

```
SegmentDirective ::= segment SegmentName [ GregName ] ";"
SegmentName ::= data | code
```

Generates a `LOC` directive for one of MMIX's conventional base locations:

- `segment data;` → `LOC #2000000000000000`
- `segment code;` → `LOC #100`

For ordinary programs, **segment data;** and **segment code;** are preferred because they say what the source means instead of exposing a magic hex constant.

Capturing the segment base address. The optional *GregName* on **segment data** allocates a GREG immediately after the LOC, capturing the data-segment base address in the same statement. This eliminates the near-universal boilerplate of a bare `greg $DataBase;` on the following line.

```
segment data $DataBase;    # LOC #2000000000000000 + DataBase GREG @
octa Table 0, 1, 2;

segment code;
```

Without the optional name the behaviour is unchanged: **segment data;** emits only the LOC.

6.4 POOL – emit a literal pool

```
PoolDirective ::= pool ";"
```

The bootstrap compiler treats **pool;** as its own directive rather than as a **segment** variant. It emits a literal-pool flush marker at that point in the instruction stream.

Procedure placement. **def** blocks declared inside the data segment (between `segment data;` and `segment code;`) are automatically relocated to the code segment by the compiler. They are emitted immediately after the `segment code;` line and before the implicit main body, so that PUSHJ instructions can reach them with a normal relative branch. No manual reordering of source code is required.

6.5 LOC – set location counter*

```
LocDirective ::= loc ConstExpr ";"
```

Generates a LOC assembler directive with an arbitrary constant address. Use it for non-standard placement, hand-written alignment tricks, or experiments with unusual MMIX layouts. For the conventional two-segment layout, prefer **segment data;** and **segment code;**.

6.6 RESERVE – reserve uninitialised space

```
ReserveDirective ::= reserve [ Label ] ConstExpr ";"
```

Advances the location counter by *ConstExpr* bytes without emitting any data into the object file. Generates `LOC @+N` in MMIXAL. Because the MMIX simulator zero-initialises all of `Data_Segment`, this is the correct idiom for allocating large zero-filled arrays. *ConstExpr* may reference named constants defined by **const**.

The optional *Label* anchors the buffer's start address: it emits `Label IS @` in MMIXAL before the LOC, giving the label the value of the current location counter. This is a compact alternative to the two-statement **greg**+**reserve** pattern when a GREG is not otherwise needed.

```
reserve Xbuf SIZE;      # Xbuf IS @; LOC @+512   (SIZE = 512)
reserve 40000;          # anonymous: LOC @+40000
```

Contrast with **byte**/**wyde**/**tetra**/**octa**: those directives emit bytes into the object file and require every value to be listed explicitly. Use a direct data-size keyword when the initial contents matter; use **reserve** when you simply need a block of zero-filled working space.

The `greg[N]` shorthand. When the base address is also needed as a register, prefer the combined `greg $Name[N]` form (see the **`greg`** section). The explicit two-statement form is still accepted and produces identical output:

```
segment data;
greg $_x[40000];   # $_x = @; LOC @+40000  (preferred)
greg $_y[40000];   # $_y = @; LOC @+40000
segment code;
```

Use the standalone **`reserve`** (with a label) when you need the address only as an assembler constant rather than a register — for example, for a **`greg`** initialised with a label-relative expression such as `greg $Xn = Xbuf + 54*8;`.

7 Namespace and Composition Directives

"Good fences make good neighbors."
— Robert Frost, *Mending Wall* (1914)

7.1 PREFIX – assembler namespace

```
PrefixDirective ::= prefix Label ";"
| prefix ";"
```

Emits a `PREFIX` assembler directive, giving C++-style namespacing. `prefix Foo;` causes subsequent labels to be assembled as `Foo:label`. `prefix;` resets to the root namespace.

7.2 LOCAL – assert a local register

```
LocalDecl ::= local Register ";"
```

Emits `LOCAL $x`, asserting that register `$x` is a local temporary, ensuring the assembler sets `$rL` high enough to cover it.

The compiler does *not* auto-emit `LOCAL`; MMIXAL infers the local register count by scanning the generated code. Explicit `local` declarations are only needed after `save`/`unsave` sequences that may alter `$rL` in ways MMIXAL cannot statically determine.

7.3 INCLUDE – source composition

```
IncludeDirective ::= include StringLit ";"
```

Includes another PL/MMIX source file into the current translation unit. The directive is resolved by the *tangler* before ordinary compilation, so the compiler proper still sees one flattened input file; feeding a `.plw` source containing `include` directives directly to the compiler, bypassing the tangler, is not supported.

The directive must occupy an entire code line on its own (optional leading whitespace and an optional trailing `#` comment are permitted, but no other tokens may share the line). The path is a double-quoted *StringLit*; the trailing `";"` is required.

Path resolution. A path beginning with `/` or `\`, or whose second character is `:` (a Windows drive letter), is treated as absolute and used verbatim. Any other path is resolved *relative to the directory of the file that contains the* `include` *directive*, not relative to the current working directory. There is no separate include search path. Backslashes encountered in paths are normalized to forward slashes when the resolved path is recorded.

Duplicate and cyclic includes. Each source file is loaded at most once per build, keyed by its resolved path. A second `include` of an already-loaded file is silently ignored, so diamond-shaped include graphs are harmless. By contrast, a `include` that points (directly or transitively) at a file currently being processed on the include stack is a *cycle* and is reported as a hard error. The tangler also enforces a maximum include nesting depth (currently 8) and an overall cap on the number of distinct included files (currently 256).

Note

This is the preferred way to share reusable code in PL/MMIX. The standard MMIX toolchain (`mmixal` + `mmix`) does not provide the sort of multi-file linker workflow needed for traditional external linkage. Consequently, library reuse should be expressed as source composition, typically handled by the tangler and mirrored by the weaver for documentation.

8 Statements

"And God said, Let there be light: and there was light."
— *Genesis* 1:3

8.1 Assignment

```
Assignment ::= RegOperand "=" Expression ";"

Expression ::= UnaryOp RegOperand    (unary operations)
| RegOperand BinaryOp RegOperand    (triadic, reg/reg)
| RegOperand BinaryOp SmallImm    (triadic, reg/imm)
| RegOperand    (register copy:  OR $x,$y,0)
| [ "-" ] Immediate    (wide constant load; "-" for negatives)
| Label    (const-alias load)
| ConstExpr    (pure compile-time constant expression)
```

RegOperand is either a register `$N` or a `$`-prefixed alias name previously declared with `register $x = $N;` (see Section 4). A bare *RegOperand* on the right-hand side copies the register (emits `OR $x,$y,0`). A bare *Immediate*, `const`-alias name, or pure *ConstExpr* loads a 64-bit constant using the minimal `SETH`/`INC*` sequence after compile-time folding. Thus `$j = m - 1;` is valid, but the triadic form remains unchanged: `$x = $y + m - 1;` is still not a single assignment expression in the language.

8.1.1 Binary operators

Operator	MMIX opcode	Description
Signed arithmetic		
`+`	`ADD`	addition
`-`	`SUB`	subtraction
`*`	`MUL`	multiplication
`/`	`DIV`	division; remainder → `$rR`
Unsigned arithmetic		
`+u`	`ADDU`	unsigned add
`-u`	`SUBU`	unsigned subtract
`*u`	`MULU`	unsigned multiply; upper 64 bits → `$rH`
`/u`	`DIVU`	unsigned divide; remainder → `$rR`
Scaled unsigned add		
`2addu`	`2ADDU`	$\$y \cdot 2 + \$z$
`4addu`	`4ADDU`	$\$y \cdot 4 + \$z$
`8addu`	`8ADDU`	$\$y \cdot 8 + \$z$ (pointer stride)
`16addu`	`16ADDU`	$\$y \cdot 16 + \$z$
Bitwise		
`&`	`AND`	bitwise AND
`|`	`OR`	bitwise OR
`^`	`XOR`	bitwise XOR
`andn`	`ANDN`	AND-NOT: $\$y$ & $\neg\$z$
`orn`	`ORN`	OR-NOT
`nand`	`NAND`	NAND
`nor`	`NOR`	NOR
`nxor`	`NXOR`	XNOR

Operator	MMIX opcode	Description
Shifts		
`«`	`SL`	arithmetic shift left
`»`	`SR`	arithmetic shift right
`«u`	`SLU`	logical shift left
`»u`	`SRU`	logical shift right
Floating point (dot suffix)		
`+.`	`FADD`	float add
`-.`	`FSUB`	float subtract
`*.`	`FMUL`	float multiply
`/.`	`FDIV`	float divide
`%.`	`FREM`	float remainder
Named operations		
`mor`	`MOR`	matrix of bits OR
`mxor`	`MXOR`	matrix of bits XOR
`mux`	`MUX`	multiplex (uses `$rM` as mask)
`sadd`	`SADD`	sideways add: popcount($\$y$ & $\neg\$z$)
`bdif`	`BDIF`	byte difference
`wdif`	`WDIF`	wyde difference
`tdif`	`TDIF`	tetra difference
`odif`	`ODIF`	octa difference
`cmp`	`CMP`	signed compare $\to \{-1, 0, 1\}$
`cmpu`	`CMPU`	unsigned compare $\to \{-1, 0, 1\}$
`cmpe.`	`FCMPE`	float compare within `$rE` neighbourhood
`eqle.`	`FEQLE`	float equivalent w.r.t. `$rE`
`une.`	`FUNE`	float unordered w.r.t. `$rE`

8.1.2 Unary operators

Operator	MMIX opcode	Description
`-`	`NEG`	negate (signed)
`-u`	`NEGU`	negate (unsigned)
`sqrt.[R]`	`FSQRT`	float square root
`int.[R]`	`FINT`	round float to integer
`flot.[R]`	`FLOT`	signed int $\to$ float
`flotu.[R]`	`FLOTU`	unsigned int $\to$ float
`fix.[R]`	`FIX`	float $\to$ signed int
`fixu.[R]`	`FIXU`	float $\to$ unsigned int
`sflot.[R]`	`SFLOT`	signed int $\to$ 32-bit float (in top 32 bits)
`sflotu.[R]`	`SFLOTU`	unsigned int $\to$ 32-bit float (in top 32 bits)

The optional `[R]` is a rounding-mode override; see the rounding discussion immediately below. Omitting it uses the current `$rA` mode.

8.1.3 Rounding mode

```
RoundStmt ::= rounding RoundMode ";"

RoundMode ::= nearest   (ROUND_NEAR - $rA bits 00; Y=4)
| zero   (ROUND_OFF - $rA bits 01; Y=1 - truncate)
| up   (ROUND_UP - $rA bits 10; Y=2 - toward +∞)
| down   (ROUND_DOWN - $rA bits 11; Y=3 - toward −∞)
```

`rounding X;` sets bits 16–17 of `$rA`. Individual floating-point unary operations can override the mode with an inline `[RoundMode]`; the global rounding mode and per-operation overrides are equivalent in effect.

```
register $src, $dst;    # $0, $1
rounding down;              # set global rounding mode
$dst = flot.[up] $src;      # FLOT with ROUND_UP override
```

8.2 Conditional and Zero-Set Statements (CS/ZS family)

```
CondSetStatement ::= RegOperand "=" CondVal if RegOperand is Polarity ";"
ZeroSetStatement ::= RegOperand "?=" CondVal if RegOperand is Polarity ";"

CondVal ::= RegOperand | SmallImm
Polarity ::= neg | zero | pos | odd
| nonneg | nonzero | nonpos | even
```

Polarity	CS opcode	ZS opcode
neg	CSN	ZSN
zero	CSZ	ZSZ
pos	CSP	ZSP
odd	CSOD	ZSOD
nonneg	CSNN	ZSNN
nonzero	CSNZ	ZSNZ
nonpos	CSNP	ZSNP
even	CSEV	ZSEV

`=` (CS) sets `$x` to *CondVal* if the condition holds, otherwise leaves `$x` unchanged. `?=` (ZS) sets `$x` to *CondVal* if the condition holds, otherwise sets `$x` to zero.

```
register $result = $2;  register $src = $3;  register $cond = $4;
$result = $src if $cond is neg;    # CSN  $2,$4,$3

register $mask = $5;  register $test = $6;
$mask ?= #ff if $test is nonzero;    # ZSNZ $5,$6,#ff -- mask=0xff if
    test!=0, else 0
```

8.3 Memory Access

```
LoadStatement ::= Register "=" "[" Address "]" SizeSuffix ";"
| Register "=" "[" Address "]" ".unc" ";"   (LDUNC - uncached)

| Register "=" Register "[" AddrIndex "]" SizeSuffix ";"   (indexed shorthand load)
| Register "=" Register "[" AddrIndex "]" ".unc" ";"   (indexed shorthand LDUNC)

| Register "=" Label "[" [ SmallImm ] "]" SizeSuffix ";"   (label-based load)
| Register "=" Label "[" [ SmallImm ] "]" ".unc" ";"   (label-based LDUNC)
```

```
StoreStatement ::= "[" Address "]" SizeSuffix "=" StoreRhs ";"
| "[" Address "]" ".unc" "=" Register ";"   (STUNC)
| "[" Address "]" ".o" "=" StoreImm ";"   (STCO - store constant)
| Register "[" AddrIndex "]" SizeSuffix "=" StoreRhs ";"   (indexed shorthand
store)
| Register "[" AddrIndex "]" ".unc" "=" Register ";"   (indexed shorthand STUNC)
| Register "[" AddrIndex "]" ".o" "=" StoreImm ";"   (indexed shorthand STCO)
| Label "[" [ SmallImm ] "]" SizeSuffix "=" StoreRhs ";"   (label-based store)
| Label "[" [ SmallImm ] "]" ".unc" "=" Register ";"   (label-based STUNC)
| Label "[" [ SmallImm ] "]" ".o" "=" StoreImm ";"   (label-based STCO)
StoreRhs ::= Register | StoreImm
StoreImm ::= SmallImm | ConstAliasName | CharLiteral Address ::= Register "+"
Register
| Register "+" SmallImm
| Register
| Label   (data label; used in cswap/cache/ldvts)
| Label "+" SmallImm   (data label + byte offset; used in cswap/cache/ldvts)
AddrIndex ::= Register | SmallImm
```

Suffix	Width	Load	Store
`.b`	byte, signed	LDB	STB
`.bu`	byte, unsigned	LDBU	STBU
`.w`	wyde, signed	LDW	STW
`.wu`	wyde, unsigned	LDWU	STWU
`.t`	tetra, signed	LDT	STT
`.tu`	tetra, unsigned	LDTU	STTU
`.o`	octa (8 bytes)	LDO	STO
`.sf`	short float	LDSF	STSF
`.ht`	high-tetra	LDHT	STHT

High-tetra notes. LDHT: `$x = M_4[$y+$z] « 32` (bottom 32 bits zeroed). STHT: `M_4[$y+$z] = $x » 32`. Useful for big-endian 32-bit interchange and `sflot./sflotu.` results.

The shorthand form `$base[off].suffix` is exactly equivalent to `[$base + off].suffix`. The index is a byte offset; there is no implicit scaling by the accessed object size. For label-based addressing, empty brackets are shorthand for zero offset, so `U[].o` and `U[0].o` are equivalent.

For stores, the RHS may also be a compile-time constant alias or a character literal. Both double-quoted (`"c"`) and single-quoted (`'c'`) forms are accepted. Character stores are most useful with `.b`/`.bu`; for example, `Buf[0].b = ' ';` or `Buf[0].b = " ";` both store the ASCII code for space, and `Buf[1].b = '
n';` stores newline.

```
register $base, $idx, $val;    # $0, $1, $2
register $ptr  = $4;  register $byte = $3;
register $dst  = $5;  register $src  = $6;
register $buf  = $7;  register $ch   = $8;

$val  = [$base + $idx].o;    # LDO  $2,$0,$1
$val  = $base[$idx].o;       # same load, shorthand form
$byte = [$ptr + 8].bu;      # LDBU $3,$4,8
$byte = $ptr[8].bu;         # same load, shorthand form
[$dst].o    = $src;         # STO  $6,$5,0
$dst[0].o   = $src;         # same store, shorthand form
[$buf + 1].b = $ch;         # STB  $8,$7,1
```

```
$buf[1].b    = $ch;          # same store, shorthand form
$buf[2].b    = " ";         # STB   32,$7,2
```

8.3.1 Label-based addressing and base-register selection

When a data label is written in the `Label[offset].suffix` form, the compiler passes it unchanged to MMIXAL, which applies *Algorithm B* to choose the base register automatically: it finds the `GREG` register whose value is closest to (and not exceeding) the label's address, then uses it as the base with the appropriate byte offset.

The programmer must ensure that a suitable **greg** exists within 255 bytes of the label. MMIXAL will report a fatal error if no such register can be found.

```
segment data;
greg $DataBase;
octa U 1000;    # U is at DataBase + 0
octa V 30;      # V is at DataBase + 8
segment code;

$1 = U[0].o;      # LDO $1,U,0   (MMIXAL picks DataBase as base)
$2 = V[0].o;      # LDO $2,V,0   (MMIXAL picks DataBase as base)
```

Note

Label-based addressing is purely a compile-time convenience: the compiler emits `LDO $x,Label,N` to MMIXAL, which is identical to writing it by hand. There is no runtime overhead. This is the same mechanism MMIX assembly programs use when they write `LDO $1,U,0` directly.

8.4 Address-of Operator

```
GetaStatement ::= Register "=" "&" Label ";"
```

Loads the address of *Label* into *Register*. The compiler selects the mechanism based on where *Label* is defined:

Code-segment label (within ±262143 instructions of the statement):
Emits `GETA $x,Label` — a single PC-relative instruction. Use for branch tables, function pointers, and nearby data.

Data-segment label (any label on a **byte**/**wyde**/**tetra**/**octa** declaration inside a **segment data** block):
The compiler automatically generates a `GREG @` entry immediately before the data declaration (capturing the current location counter) and emits `SET $x,__G_Label` in the code (the auto-GREG is named `__G_Label`). Multiple `&Label` references share one GREG. The auto-GREG is visible in the generated `.mms` file.

In both cases the programmer writes only `$r = &Label;`.

```
# Code-segment label: GETA
register $tbl = $0;
$tbl = &MyTable;    # GETA $0,MyTable

# Data-segment label: auto-GREG + SET (no explicit greg declaration
    needed)
segment data $DataBase;
byte ErrMsg "error\n";
```

```
segment code;
register $addr, $len;
$addr = &ErrMsg;    # SET $addr,__G_ErrMsg  (GREG @ emitted before
    ErrMsg data)
$len  = 6;
trap 0, FPUTS, STDOUT;
```

8.5 Wyde-Immediate Bit Updates

```
WydeSetStatement ::= Register WydeOp WydeImm ";"
WydeOp ::= orh=   (ORH - $x |= (YZ « 48))
| ormh=   (ORMH - $x |= (YZ « 32))
| orml=   (ORML - $x |= (YZ « 16))
| orl=   (ORL - $x |= YZ)
| andnh=   (ANDNH - $x &= ~(YZ«48))
| andnmh=   (ANDNMH - $x &= ~(YZ«32))
| andnml=   (ANDNML - $x &= ~(YZ«16))
| andnl=   (ANDNL - $x &= ~YZ)
```

Note

Division of responsibilities between assignment and wyde ops.
Constant loading — use the plain assignment form (`$x = Immediate;`). The compiler automatically emits the minimal SETH/SETMH/SETML/SETL + OR* sequence. Never write wyde ops merely to load a constant.
Bit-field manipulation — use `WydeSetStatement` when you need to selectively OR bits into, or AND bits out of, one 16-bit field of a register that already holds other data. These are operations MMIX provides directly (ORH, ORMH, ANDNH, ...) and they have no single-instruction equivalent through the assignment form.

```
register $x = $0;
# Build #DEADBEEFCAFEBABE in $x step by step
$x = 0;
$x orh=  #DEAD;    # ORH  -- $x = #DEAD_0000_0000_0000
$x ormh= #BEEF;    # ORMH -- $x = #DEAD_BEEF_0000_0000
$x orml= #CAFE;    # ORML -- $x = #DEAD_BEEF_CAFE_0000
$x orl=  #BABE;    # ORL  -- $x = #DEAD_BEEF_CAFE_BABE

# Compute |f| for a 64-bit float: clear sign bit (bit 63)
register $f = $1;
$f andnh= #8000;    # ANDNH $1,#8000
```

9 Special Registers and Predefined Constants

"In its proper place, every thing is good; out of its place, every thing is bad."
— Antoine Furetière, *Dictionnaire universel* (1690)

MMIX has 32 *special registers* accessed with the `GET` and `PUT` instructions. PL/MMIX exposes them with the same regular assignment syntax used for general registers; the compiler chooses GET or PUT based on which side of the `=` the special name appears. Alongside the registers themselves, the compiler predefines a small set of named constants for status bits, fence modes, and OS trap codes. All of these names can be shadowed by a user-defined `const` declaration if a program needs a private value with the same spelling.

9.1 GET and PUT

```
SpecialRegAssign ::= UserSpecialReg "=" Register ";"    (PUT)
| PrivSpecialReg "=" Register ";"    (PUT, privileged)
| Register "=" AnySpecialReg ";"    (GET)
```

```
$1 = $rR;    # GET $1,rR -- read remainder after DIV
$rJ = $0;    # PUT rJ,$0 -- save return address
$2 = $rK;    # GET $2,rK -- read interrupt mask
$rK = $3;    # PUT rK,$3 -- set interrupt mask (privileged)
```

9.2 Register inventory

The 32 special registers fall into three tiers based on privilege.

Tier 1 – user GET and PUT

Name	Description
`$rA`	arithmetic status register
`$rD`	dividend register
`$rE`	epsilon register
`$rG`	global threshold
`$rH`	himult register
`$rJ`	return-jump register
`$rM`	multiplex mask
`$rR`	remainder register

Tier 2 – user GET only

Name	Description
`$rF`	failure location register
`$rI`	interval counter
`$rL`	local threshold
`$rN`	serial number
`$rU`	usage counter

Tier 3 – GET by all; PUT requires privilege

Attempting PUT from user mode (when the relevant bit of `$rK` is not cleared) raises a privileged-instruction interrupt.

Name	Description
`$rB`	bootstrap register (trip)
`$rBB`	bootstrap register (trap)
`$rC`	continuation register
`$rK`	interrupt mask register
`$rO`	register stack offset
`$rP`	prediction register (CSWAP)
`$rQ`	interrupt request register
`$rS`	register stack pointer
`$rT`	trap address register
`$rTT`	dynamic trap address register
`$rV`	virtual translation register
`$rW`	where-interrupted register (trip)
`$rWW`	where-interrupted register (trap)
`$rX`	execution register (trip)
`$rXX`	execution register (trap)
`$rY`	Y operand (trip)
`$rYY`	Y operand (trap)
`$rZ`	Z operand (trip)
`$rZZ`	Z operand (trap)

9.3 Predefined constants

Arithmetic-exception flag bits (`$rA`)

Name	Value	$rA bit	Meaning
`$rA_D`	`#80`	7	integer **D**ivide exception
`$rA_V`	`#40`	6	float in**V**alid operation
`$rA_W`	`#20`	5	float o**W**erflow
`$rA_I`	`#10`	4	float **I**nexact
`$rA_O`	`#08`	3	integer **O**verflow
`$rA_U`	`#04`	2	float **U**nderflow
`$rA_Z`	`#02`	1	float **Z**ero-divide
`$rA_X`	`#01`	0	float ine**X**act (enable)

These constants are most useful when reading or writing `$rA`, and as the exception-type argument to `trip` (see Section 14.2):

```
# Enable divide and overflow exceptions in $rA
$1 = $rA;              # read current exception flags
$1 = $1 | rA_D;      # set Divide enable bit
$1 = $1 | rA_O;      # set Overflow enable bit
$rA = $1;              # write back

# Manually raise a divide-by-zero trip
trip rA_D, 0, 0;     # TRIP rA_D,0,0
```

Rounding mode

Name	Value	Meaning
`ROUND_CURRENT`	0	use current rounding mode from `$rA`

The values 1–4 are spelled with the keywords `zero`, `up`, `down`, and `nearest` in floating-point statements (see Section 8.1.3). `ROUND_CURRENT` is the only rounding value that is a predefined

constant rather than a keyword.

Memory-fence argument (`sync`)

Name	Value	Effect
`SYNC_DRAIN`	`0`	drain pipeline
`SYNC_STORES`	`1`	store-store fence
`SYNC_LOADS`	`2`	load-load fence
`SYNC_ALL`	`3`	full load/store fence

Values 4–7 are privileged (power-saver, cache flush, TLB invalidate) and have no predefined names.

Resume-from-interrupt mode (`resume`)

Name	Value	Meaning
`RESUME_REEXECUTE`	`0`	re-execute instruction in `$rW`
`RESUME_CONTINUE`	`1`	resume with substituted Y, Z operands
`RESUME_SET`	`2`	deposit `$rZ` into named register, resume
`RESUME_PAGEFILL`	`3`	OS page-table fill (install `$rZ` as PTE)

NNIX simulator trap codes

The NNIX simulator interprets `trap 0, Y, Z;` according to the following routine indices (Y) and stream identifiers (Z):

Name	Value	Routine
`HALT`	0	terminate process
`FOPEN`	1	open file
`FCLOSE`	2	close file
`FREAD`	3	read from file
`FGETS`	4	read string (byte)
`FGETWS`	5	read string (wyde)
`FWRITE`	6	write to file
`FPUTS`	7	write string (byte)
`FPUTWS`	8	write string (wyde)
`FSEEK`	9	seek in file
`FTELL`	10	report file position
`STDIN`	0	standard input stream
`STDOUT`	1	standard output stream
`STDERR`	2	standard error stream

```
trap 0, FPUTS,  STDOUT;   # write string to stdout
trap 0, FPUTS,  STDERR;   # write string to stderr
trap 0, HALT,   0;        # terminate process
```

10 Control Flow

"Would you tell me, please, which way I ought to go from here?
—That depends a good deal on where you want to get to."
— Lewis Carroll, *Alice's Adventures in Wonderland* (1865)

10.1 If statement

```
IfStatement ::= if [ BranchHint ] Condition ":" StatementList
                { elif [ BranchHint ] Condition ":" StatementList }
                [ else ":" StatementList ]
                end
```

```
register $x, $n;    # $0, $1
if $x > 0:
  $n = $n + 1;
else:
  $n = $n - 1;
end
```

Any number of **elif** clauses may appear between the first arm and the optional **else**. Each **elif** is simply shorthand for a nested **else**: **if** ... **end**; the generated MMIXAL is the same branch chain you would write by hand.

```
if $x < 0:
  $sign = -1;
elif $x == 0:
  $sign = 0;
else:
  $sign = 1;
end
```

10.2 Postfix if

Any statement that ends with **;** (except **CondSet** and **ZeroSet**, which use **if** internally) may be written in postfix conditional form: the **if** keyword and a condition are inserted *before* the trailing semicolon.

```
PostfixIfStatement ::= SimpleStatement if [ BranchHint ] Condition ";"
```

The statement is executed only when the condition is true. This is equivalent to the block form

```
if [BranchHint] Condition:
  SimpleStatement
end
```

but is more compact for single-statement guards. The optional **likely**/**unlikely** hint works exactly as in the block form, selecting the PB* or B* instruction.

```
goto Loop if n is nonzero;              # conditional branch
return    if k is zero;                 # early exit
$x = $y + $z if $a != 0;               # conditional assignment
break     if done is nonzero;           # conditional loop exit
goto Loop if likely n is nonzero;      # PB* -- loop usually taken
```

10.3 While statement

```
WhileStatement ::= while [ BranchHint ] Condition ":" StatementList end
```

```
register $i = $1;  register $sum = $2;
while $i != 0:
  $sum = $sum + $i;
  $i   = $i -u 1;
end
```

A larger example: Euclid's algorithm for the greatest common divisor. The remainder of the signed division goes to `$rR`, so a single `while` loop suffices.

```
def gcd($a=$0, $b=$1):
  register $t = $2;
  while $b != 0:
    $t = $a / $b;    # DIV -- quotient in $t, remainder in $rR
    $a = $b;
    $b = $rR;        # GET remainder
  end
  return $a;
end

register $m, $n = $4, $result;
$result = call gcd($m, $n);
```

10.4 Loop statement

```
LoopStatement ::= loop ":" StatementList end
```

`loop` is the structured form of the classic "top label plus back-edge jump" idiom. The compiler emits one loop-top label, the body, and an unconditional `JMP` back to that label. It is most useful when the natural exits occur in the middle of the body rather than at the header.

```
loop:
  $ch = [$src].bu;
  break if $ch is zero;
  [$dst].bu = $ch;
  $src = $src +u 1;
  $dst = $dst +u 1;
end
```

10.5 Do. . . while statement

```
DoWhileStatement ::= do ":" StatementList end while [ BranchHint ] Condition ";"
```

The body executes *unconditionally* on the first pass. At the bottom (`end while`), if the condition is true the compiler branches back to the top; otherwise execution falls through. This saves one instruction compared to `while`: no `JMP` at the bottom and no end label.

The `end` closes the body (consistent with `if` and `while`); the `while` clause immediately after it specifies the back-edge condition.

```
register $i = $1;  register $sum = $2;
$i = 10;
```

```
do:
  $sum = $sum + $i;
  $i   = $i -u 1;
end while $i != 0;
```

Use **do**...**while** whenever the loop body must execute at least once and the count is set before entering the loop. Knuth's digit-extraction routines and permutation inversions are natural fits.

10.6 Break and continue

```
BreakStatement ::= break ";"
ContinueStatement ::= continue ";"
```

These statements are valid only inside **while**, **do**...**while**, or **loop**. They are deliberately thin abstractions:

- **break** lowers to one jump to the current loop's end label.
- **continue** lowers to one jump to the current loop's continue target. For **while** and **loop**, that target is the loop header; for **do**...**while**, it is the bottom-of-loop condition test.

Because they are ordinary semicolon-terminated simple statements, they also work with postfix **if**:

```
while $n != 0:
  $n = $n -u 1;
  continue if $n > 3;
  break if $n is zero;
end
```

10.7 Branch hints

```
BranchHint ::= likely   (emit PB* - branch usually taken)
| unlikely   (emit B* - branch usually not taken)
```

Without a hint, a standard `B*` instruction is emitted (same as **unlikely**). The keyword **unlikely** is purely documentary. **likely** selects the `PB*` (probable-branch) variant, which on some MMIX implementations costs 3v when correctly predicted rather than nothing.

```
register $ptr, $n, $sum, $x;    # $0-$3
while likely $n != 0:    # PBNZ -- loop usually continues
  $x   = [$ptr].o;
  $sum = $sum + $x;
  $ptr = $ptr +u 8;
  $n   = $n   -u 1;
end
```

10.8 Conditions

```
Condition ::= RegOperand CompareOp RegOperand
| RegOperand CompareOp SmallImm
| RegOperand FloatCompareOp RegOperand
| RegOperand is Polarity

CompareOp ::= "==" | "!="
| "<" | "<=" | ">" | ">="   (signed, via CMP)
```

```
| "<u" | "<=u"| ">u" | ">=u"    (unsigned, via CMPU)
FloatCompareOp ::= "==." | "!=." | "<." | "<=." | ">." | ">=."
```

Integer conditions. The compiler emits `CMP`/`CMPU` into a fresh temporary, then branches on it with the appropriate `B*`/`PB*` instruction.

Float conditions. MMIX has no float branch instructions; the compiler emits the corresponding float comparison into a temporary, then branches with `BNZ`/`BZ`.

`is` polarity. The compiler emits a direct `BN`/`PBP`/etc. on the register's value—no CMP emitted. See Section 8.1 for the *Polarity* table.

Lowering table. The complete lowering of every condition form to MMIX instructions is given below. Each row shows what the compiler emits for a **`goto`** guarded by the condition; **`while`**, postfix **`if`**, and the block **`if`** all use the same sequence (the surrounding control structure may invert the polarity, in which case the dual opcode in the same row is emitted).

Source form	Compare	Branch (taken)	Branch (not taken)
`$x is zero`	—	`BZ`	`BNZ`
`$x is nonzero`	—	`BNZ`	`BZ`
`$x is neg`	—	`BN`	`BNN`
`$x is nonneg`	—	`BNN`	`BN`
`$x is pos`	—	`BP`	`BNP`
`$x is nonpos`	—	`BNP`	`BP`
`$x is odd`	—	`BOD`	`BEV`
`$x is even`	—	`BEV`	`BOD`
`$x == $y`	`CMP`	`BZ`	`BNZ`
`$x != $y`	`CMP`	`BNZ`	`BZ`
`$x < $y`	`CMP`	`BN`	`BNN`
`$x <= $y`	`CMP`	`BNP`	`BP`
`$x > $y`	`CMP`	`BP`	`BNP`
`$x >= $y`	`CMP`	`BNN`	`BN`
`$x <u $y`	`CMPU`	`BN`	`BNN`
`$x <=u $y`	`CMPU`	`BNP`	`BP`
`$x >u $y`	`CMPU`	`BP`	`BNP`
`$x >=u $y`	`CMPU`	`BNN`	`BN`
`$x ==. $y`	`FEQL`	`BNZ`	`BZ`
`$x !=. $y`	`FEQL`	`BZ`	`BNZ`
`$x <. $y`	`FCMP`	`BN`	`BNN`
`$x <=. $y`	`FCMP`	`BNP`	`BP`
`$x >. $y`	`FCMP`	`BP`	`BNP`
`$x >=. $y`	`FCMP`	`BNN`	`BN`

When a **`likely`** hint is present, the corresponding `PB*` form is emitted in place of `B*` (`PBZ`, `PBNZ`, `PBN`, `PBNN`, `PBP`, `PBNP`, `PBOD`, `PBEV`). **`unlikely`** is documentary only and produces the same code as no hint.

Postfix `if` on `goto`. The combination `goto L if` *Cond*`;` is lowered as a single *taken* branch in the table above (no extra unconditional jump is inserted). This is the most common postfix form and the one the bootstrap parser/codegen path is most carefully tested against. Other postfix forms (`return`, assignment, **`break`**, **`continue`**) are lowered as `branch-over-fallthrough`: the compiler emits the *not-taken* branch in the table above, jumps over the guarded statement, then emits the statement at the fall-through label.

10.9 Goto statement

```
GotoStatement ::= goto Label ";"    (JMP)
| goto "[" Register "]" ";"    (GO $tmp,$r,0 - indirect)
```

Indirect goto: `$tmp` receives $\lambda + 4$ (the address of the instruction following `GO`), which is normally discarded.

```
goto Done;    # JMP Done
register $fptr = $3;
goto [$fptr];    # GO $tmp,$3,0
```

11 Cache and Memory-Order Statements*

"Great is this force of memory, excessive great, O my God;
a large and boundless chamber! Who ever sounded the bottom thereof?"
— Saint Augustine, *Confessions* X.8 (c. 400 AD)

11.1 Cache prefetch and writeback

```
CacheStatement ::= preld SmallImm "," Address ";"   (PRELD - prefetch X+1 bytes
into data cache)
| prego SmallImm "," Address ";"   (PREGO - prefetch into instruction cache)
| prest SmallImm "," Address ";"   (PREST - prestore:  signal bytes will be
written)
| syncd SmallImm "," Address ";"   (SYNCD - flush X+1 bytes to memory)
| syncid SmallImm "," Address ";"   (SYNCID - sync instruction and data caches)
```

11.2 Memory fence

```
SyncStatement ::= sync SmallImm ";"
```

The *SmallImm* argument selects the fence operation:

Value	Predefined name	Effect
0	`SYNC_DRAIN`	drain pipeline (all preceding instructions complete)
1	`SYNC_STORES`	all stores before SYNC committed before stores after
2	`SYNC_LOADS`	all loads before SYNC committed before loads after
3	`SYNC_ALL`	all loads/stores before SYNC committed before any after
4–7	—	privileged: power-saver, cache flush, TLB invalidate

The predefined constant aliases `SYNC_DRAIN`, `SYNC_STORES`, `SYNC_LOADS`, and `SYNC_ALL` are described in Section 9.

```
register $buf = $5;
# Publish 64 bytes of shared buffer at $buf
syncd 63, $buf;            # SYNCD 63,$5,0 -- flush 64 bytes
sync SYNC_STORES;          # SYNC 1 -- store-store ordering
```

12 Compare-and-Swap*

“If he be Mr. Hyde… I shall be Mr. Seek.”
— Robert Louis Stevenson, *Strange Case of Dr Jekyll and Mr Hyde* (1886)

```
CswapStatement ::= Register "=" cswap "[" Address "]" ";"
```

Emits `CSWAP $x,$y,$z`. Atomic semantics:

$$\text{if } M_8[\$y+\$z] = \texttt{\$rP} \text{ then } M_8[\$y+\$z] = \$x;\ \$x = 1 \text{ else } \texttt{\$rP} = M_8[\$y+\$z];\ \$x = 0$$

Set `$rP` to the expected value before the instruction.

```
register $addr, $old, $new, $ok;   # $0-$3
# Lock-free counter increment at address $addr
retry:
  $old = [$addr].o;        # LDO -- load current value
  $rP  = $old;             # PUT $rP -- expected value for CAS
  $new = $old + 1;         # ADD -- desired new value
  $ok  = cswap [$addr];    # CSWAP $3,$0,0 -- attempt swap
  if $ok == 0:
    goto retry;
  end
```

13 Register-Stack Save and Restore*

"I'll be back."
— Arnold Schwarzenegger, *The Terminator* (1984)

```
SaveStatement ::= save Register ";"   (SAVE $x,0)
| unsave Register ";"   (UNSAVE 0,$z)
```

save snapshots the entire register stack to memory. `$x` must be a global register; after SAVE, `$x` holds the address of the saved frame. **unsave** restores from a previously saved frame. Used for coroutine and task switching.

14 Trips, Traps, and Resume*

“TRIP, TRAP! TRIP, TRAP! went the bridge.
‘WHO’S THAT TRIPPING OVER MY BRIDGE?’ roared the troll.”
— P. C. Asbjørnsen & J. Moe, *Three Billy Goats Gruff* (1841)

14.1 Trap (OS service request)

```
TrapStatement ::= trap SmallImm "," SmallImm "," SmallImm ";"
```

Emits `TRAP X,Y,Z`. The three byte fields are available in `$rXX` for the OS to inspect.

Predefined MMIX conventions:

Instruction	Effect
`trap 0,0,0`	terminate process
`trap 0,1,0`	OS default exception handling

NNIX simulator: `Y` selects the service routine (FOPEN, FPUTS, …), `Z` selects the I/O stream.

```
# Hello-world via NNIX traps.
segment data;
greg $IOBase;
byte Hello "Hello\n";

segment code;
trap 0, FPUTS, STDOUT;    # print Hello\n
trap 0, HALT,  0;
```

Installing a custom trap handler (privileged). A kernel that wants to handle its own traps writes the interrupt-mask register `$rK` and points the dynamic trap-address register `$rT` at its handler routine. Both PUTs require privileged mode.

```
register $mask    = $1;
register $handler = $2;

# Enable all machine-level interrupt bits.
$mask = $rK;              # GET $1,15
$mask = $mask | #ff;
$rK   = $mask;            # PUT 15,$1 -- privileged

# Install custom trap handler.
$handler = &MyHandler;  # GETA -- load handler address
$rT      = $handler;    # PUT 13,$2 -- set trap vector
```

14.2 Trip (user exception)

```
TripStatement ::= trip SmallImm "," SmallImm "," SmallImm ";"
```

Emits `TRIP X,Y,Z`. Saves `$255` to `$rB`, sets `$rJ = 0`, stores the instruction in `$rX`, puts the values of `$Y` and `$Z` in `$rY` and `$rZ`, and jumps to virtual address 0. Arithmetic exception handlers start at addresses 16, 32, …, 128 for the DVWIOUZX bits of `$rA`.

14.3 Resume

```
ResumeStatement ::= resume ";"   (RESUME 0)
| resume SmallImm ";"
```

Z	Predefined name	Effect
0	`RESUME_REEXECUTE`	re-execute instruction in `$rW` with operands from `$rY`, `$rZ`
1	`RESUME_CONTINUE`	re-execute with `$rY`, `$rZ` substituted as Y and Z operands
2	`RESUME_SET`	deposit `$rZ` into register named by bits 16–23 of `$rX`, then resume
3	`RESUME_PAGEFILL`	OS page-table fill: install `$rZ` as PTE for the page in `$rY`

`resume;` is shorthand for `resume 0;`.

PTE (Page Table Entry). A PTE is the hardware data structure that describes one virtual memory page. On MMIX it is a 64-bit word stored in the page table; its fields encode the physical page frame number, protection bits (readable, writable, executable), and a validity bit. When a TLB miss occurs, the OS trap handler walks the page table, finds the PTE for the faulting page, and installs it into the TLB with `resume RESUME_PAGEFILL;` (`RESUME 3`), which atomically loads the PTE from `$rZ` and restarts the faulting instruction.

15 Translation Lookaside Buffer Management (LDVTS)*

"The map is not the territory."
— Alfred Korzybski, *Science and Sanity* (1933)

The *translation lookaside buffer* (TLB) is a hardware cache of recent virtual-to-physical address translations. Every memory access goes through the TLB; a *TLB miss* triggers a hardware or software page-table walk. The `ldvts` instruction is the privileged interface for updating TLB entries from OS code.

```
LdvtsStatement ::= ldvts Register "," Address ";"
```

Emits `LDVTS $x,$y,$z`. Updates the TLB entry for the page containing virtual address `$y+$z`. Privileged. Bits of `$x`:

- bit 0: "present" – page is mapped
- bit 1: "recently used" – hint to replacement policy

If the address is not in the TLB the instruction has no effect.

16 Literate Programming with `.plw` Files

"How do I know what I think until I see what I say?"
— E. M. Forster, *Aspects of the Novel* (1927)

PL/MMIX supports a CWEB-style literate programming layer that is strictly orthogonal to the language semantics. Literate constructs live in a preprocessor layer; every `.plw` file tangled into a `.plm` file feeds the standard compiler unchanged.

16.1 Tools

Invocation	Output	Purpose
`plmmix-tangle foo.plw`	`foo.plm`	expand chunks; feed compiler
`plmmix-weave foo.plw`	`foo.tex`	LaTeX literate document

`plmmix-weave` produces a complete, self-contained LaTeX document. Compile it with `pdflatex` (requires `fancyvrb`, `hyperref`, `geometry`, `microtype` — all standard in TeX Live and MikTeX).

16.2 File format

A `.plw` file alternates between prose sections and code chunks. Prose sections are raw LaTeX source: write LaTeX commands and mathematics (`$...$`, `\[...\]`) directly in prose sections. TeX does not need to be escaped; `plmmix-weave` passes prose through unchanged.

Sigil	Meaning
`@*`	major headed section (raw LaTeX prose)
`@`	minor prose section (no heading)
`@<name@>=`	define a named code chunk
`@<name@>`	reference a chunk (inside code)
`@@`	literal `@` in prose or code

16.3 Named chunks

Chunks may be extended by defining `@<name@>=` more than once; subsequent definitions are appended. The root chunk `@<Program@>=` is the entry point for `plmmix-tangle`. Chunk names are case-sensitive; internal whitespace is normalised to a single space.

16.4 Comment conventions

Within `.plw` files, PL/MMIX code follows the standard comment syntax:

Form	Behaviour
`# ...`	line comment – kept by tangle, shown in weave

Prose sections in `.plw` files are raw LaTeX and may contain any valid LaTeX commands and comments.

16.5 #line directives

`plmmix-tangle` emits a `#line` directive before each chunk expansion so that errors are mapped back to the original `.plw` file:

```
#line "Chunk name" "foo.plw" 42
```

The compiler accepts these directives automatically, so a tangled `.plm` file can be passed straight to `plmmix-compiler` without any extra flag. If you prefer generated source without them, `plmmix-tangle` supports `-no-line` to suppress them in the tangled output. Nested chunk expansions each receive their own `#line`; a restoring directive follows each nested expansion.

16.6 Weave output

`plmmix-weave` produces a LaTeX document in the spirit of CWEB's *cweave*:

- Prose sections pass through as raw LaTeX (including math).
- Code chunks are typeset with CWEB-style syntax formatting:
  - Language keywords (`if`, `while`, `greg`, etc.) appear in **bold typewriter**.
  - Register names (`$count`, `$255`, etc.) appear in *italic typewriter*.
  - Line comments (`# ...`) appear in *italic*.
  - All other tokens (labels, numbers, operators) are in regular typewriter, with `$`, `#`, and other PL/MMIX characters printing literally.
- Chunk names in code are typeset as ⟨*name*⟩ with hyperlinks to the first definition.
- First definitions are headed ⟨*name*⟩ ≡; continuations are headed ⟨*name*⟩ (part *N*) +≡.
- `@*` sections produce numbered `\section`s; `@` sections produce unnumbered prose paragraphs.

16.7 Cross-reference index

`plmmix-weave` appends a cross-reference index (as LaTeX `description` lists) containing:

- **Chunks**: the section(s) where each chunk is defined and every section that references it, all as `hyperref` links.
- **Labels**: first definition (`[def]`) and all uses.
- **Registers**: all sections where each `$N` appears (useful for data-flow tracing across chunks).

16.8 Complete example: TPK algorithm in `.plw`

Below is the complete source of `examples/tpk.plw`, a literate program that implements the classic TPK algorithm for floating-point testing. The program reads an array of 11 numbers, computes $y = \sqrt{|t|} + 5t^3$ for each element t, and prints the result or a "TOO LARGE" message if $y > 400$.

The `.plw` format mixes raw LaTeX prose sections (marked with `@*` for heading, `@` for text) with named code chunks (defined with `@<name@>=` and referenced with `@<name@>`). Running `plmmix-weave` on this file produces a self-contained LaTeX document; the output is shown in Appendix B.

Source code: `examples/tpk.plw`

```
@* The TPK Algorithm in PL/MMIX.

The TPK algorithm (Trabb Pardo--Knuth, 1977) is a small benchmark
   program
designed to exercise a language's essential features: input, output,
   arrays,
functions, and floating-point arithmetic.  It reads $N=11$ values
$t_0,\ldots,t_{10}$, then for each index $i$ from 10 down to 0
   computes
\[
  y = \sqrt{|t_i|} + 5t_i^3
\]
```

```
and prints either ``$i$~TOO LARGE'' if $y>400$, or ``$i$~$y$'' with
    $y$
formatted to 16 significant digits.

This PL/MMIX implementation handles two non-trivial helper procedures
explicitly: decimal floating-point input is parsed by \texttt{
    ReadNumber},
and the result is formatted by \texttt{FormatLine}; the main loop
    simply
calls them.  Supporting data is interleaved with the code in each
    section
rather than gathered in one place, keeping each datum close to the
    code
that uses it.

@* Program structure.

The MMIX convention places the data segment at address \texttt
    {\#2000000000000000}
and the code segment at \texttt{\#100}.  All data declarations
    accumulate in
\verb|@<Data segment@>|, which therefore appears first in the
    assembled file.

In the code segment the two helper procedures reside after the \verb|
    HALT|
instruction.  The main body executes first and falls into \verb|HALT|;
     the
helpers are reached only via \verb|PUSHJ|, which can reach addresses
    both
before and after the call site (within $\pm256\,\text{K}$ instructions
    ),
so placing helpers after \verb|HALT| is entirely valid.  This keeps
    the
main body as a single unbroken sequence of instructions with no helper
code interspersed.

Register \verb|$255| is MMIX's conventional trap I/O channel: \verb|
    FGETS|
deposits its result count there, and \verb|FPUTS| reads the string
    address
from it.  The alias \verb|$out| is used throughout so call sites
    remain
readable.

@<Program@>=
@<Data segment@>
segment code;
register $out = $255;
@<Read input values@>
@<Process in reverse@>
trap 0, HALT, 0;
@<Input helpers@>
@<Output helper@>

@* Reading the inputs.
```

```
The input loop calls \texttt{ReadNumber} $N$ times, storing each
    result into
successive octabytes of \texttt{InputArray}.  Register \verb|$addr|
    walks the
array; \verb|$count| counts down so the loop needs no separate index.

@<Data segment@>=
const N = 11;                           # number of input values
segment data $DataBase;
octa InputArray 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
@<TooLarge data@>
greg $count = 0;
greg $addr = 0;
greg $t = 0;

@<Read input values@>=
$addr = &InputArray;
$count = N;
while likely $count != 0:
  $t = call ReadNumber();
  [$addr].o = $t;
  $addr = $addr + 8;
  $count = $count - 1;
end

@* Computing and printing in reverse.

After the read pass \verb|$addr| points one element past the array, so
    the
reverse walk steps it back by one octabyte first.

The absolute value of $t$ is obtained by clearing the IEEE~754 sign
    bit
(\verb|andnh= #8000|) before passing the value to \verb|sqrt.|.
The cubic term is computed as $t\cdot t\cdot t\cdot 5$, accumulating
    into the
same register without a temporary.

The loop's three work registers are \verb|$i| (current index), \verb|
    $u|
(absolute value of~$t$, computed in-place before the square root), and
\verb|$y| (accumulated result).  Of the five floating-point constants,
\verb|$five_f| and \verb|$threshold_f| are specific to this algorithm;
\verb|$zero_f|, \verb|$one_f|, and \verb|$ten_f| are used mainly by
    the
helper procedures and are declared here because all global registers
    share
a single GREG pool.  All five are given literal float initialisers:
PL/MMIX folds each to its IEEE~754 bit pattern at compile time, so no
runtime \texttt{FLOT} instruction is needed.

@<Data segment@>=
greg $i = 0;
greg $u = 0;
greg $y = 0;
greg $five_f = 5.0;
greg $threshold_f = 400.0;
greg $zero_f = 0.0;
```

```
greg $one_f = 1.0;
greg $ten_f = 10.0;

@<Process in reverse@>=
$addr = $addr - 8;
$i = N - 1;
$count = N;

while likely $count != 0:
  $t = [$addr].o;

  $u = $t;
  $u andnh= #8000;              # |t|: clear sign bit
  $u = sqrt. $u;

  $y = $t *. $t;
  $y = $y *. $t;                # t^3
  $y = $y *. $five_f;           # 5t^3
  $y = $u +. $y;                # sqrt(|t|) + 5t^3

  $out = call FormatLine($i, $y);
  trap 0, FPUTS, STDOUT;

  $addr = $addr - 8;
  $i = $i - 1;
  $count = $count - 1;
end

@* The floating-point input parser.

\texttt{ReadNumber} reads one line from standard input via \verb|FGETS
    | and
parses the first decimal floating-point token on that line.  It
    recognises
an optional sign, an integer part, an optional fractional part, and an
optional decimal exponent (\texttt{e} or \texttt{E}).  Before reading
    the
number, any leading bytes with value at most 32 are skipped---this
    covers
spaces, tabs, carriage returns, and newlines alike.

The value is assembled in three stages.  First the integer and
    fractional
parts are accumulated as integers (\verb|$rd_int_part| and
\verb|$rd_frac_part|), with \verb|$rd_frac_scale| counting the
    implicit
denominator (initially 1, multiplied by 10 for each fractional digit).
Then both are converted to float and combined.  Finally the exponent
    loop
rescales by repeated multiplication or division by 10.0.

The six mutable parser state registers are globals; they are reset at
    the
top of every call.  The line buffer \verb|$InputBuf| holds up to 127
characters plus a null terminator; \verb|$line\_ptr| advances byte by
    byte
through the buffer after each \verb|FGETS|.
```

```
@<Data segment@>=
greg $line_ptr = 0;
greg $InputBuf[128];
greg $ReadArgs;
octa ReadArgsData 0, 127;
greg $rd_sign = 0;
greg $rd_int_part = 0;
greg $rd_frac_part = 0;
greg $rd_frac_scale = 1;
greg $rd_exp_sign = 0;
greg $rd_exp_val = 0;

@<Input helpers@>=
@<Read one floating-point value procedure@>

@<Read one floating-point value procedure@>=
def ReadNumber() -> $result:
  register $c = $0;
  register $tmp, $value_f, $frac_f, $scale_f;

  [$ReadArgs].o = $InputBuf;
  $out = $ReadArgs;
  trap 0, FGETS, STDIN;
  goto EndRead if unlikely $out is nonpos;
  $line_ptr = $InputBuf;

  $rd_sign = 0;
  $rd_int_part = 0;
  $rd_frac_part = 0;
  $rd_frac_scale = 1;
  $rd_exp_sign = 0;
  $rd_exp_val = 0;

  SkipSpace:
  $c = [$line_ptr].bu;
  goto EndRead if unlikely $c is zero;
  $line_ptr = $line_ptr + 1;
  $tmp = $c cmpu ' ';
  goto SkipSpace if $tmp is nonpos;

  $tmp = $c cmpu '-';
  goto CheckPlus if likely $tmp is nonzero;
  $rd_sign = 1;
  $c = [$line_ptr].bu;
  $line_ptr = $line_ptr + 1;
  goto IntegerDigits;

  CheckPlus:
  $tmp = $c cmpu '+';
  goto IntegerDigits if likely $tmp is nonzero;
  $c = [$line_ptr].bu;
  $line_ptr = $line_ptr + 1;

  IntegerDigits:
  $tmp = $c cmpu '0';
  goto AfterInteger if unlikely $tmp is neg;
  $tmp = $c cmpu '9';
  goto AfterInteger if unlikely $tmp is pos;
```

```
$rd_int_part = $rd_int_part * 10;
$tmp = $c - '0';
$rd_int_part = $rd_int_part + $tmp;
$c = [$line_ptr].bu;
$line_ptr = $line_ptr + 1;
goto IntegerDigits;

AfterInteger:
$tmp = $c cmpu '.';
goto AfterFraction if $tmp is nonzero;
$c = [$line_ptr].bu;
$line_ptr = $line_ptr + 1;

FractionDigits:
$tmp = $c cmpu '0';
goto AfterFraction if unlikely $tmp is neg;
$tmp = $c cmpu '9';
goto AfterFraction if unlikely $tmp is pos;
$rd_frac_part = $rd_frac_part * 10;
$tmp = $c - '0';
$rd_frac_part = $rd_frac_part + $tmp;
$rd_frac_scale = $rd_frac_scale * 10;
$c = [$line_ptr].bu;
$line_ptr = $line_ptr + 1;
goto FractionDigits;

AfterFraction:
$tmp = $c cmpu 'E';
goto HaveExponent if $tmp is zero;
$tmp = $c cmpu 'e';
goto HaveExponent if $tmp is zero;
goto BuildValue;

HaveExponent:
$c = [$line_ptr].bu;
$line_ptr = $line_ptr + 1;
$tmp = $c cmpu '-';
goto CheckExponentPlus if $tmp is nonzero;
$rd_exp_sign = 1;
$c = [$line_ptr].bu;
$line_ptr = $line_ptr + 1;
goto ExponentDigits;

CheckExponentPlus:
$tmp = $c cmpu '+';
goto ExponentDigits if $tmp is nonzero;
$c = [$line_ptr].bu;
$line_ptr = $line_ptr + 1;

ExponentDigits:
$tmp = $c cmpu '0';
goto BuildValue if unlikely $tmp is neg;
$tmp = $c cmpu '9';
goto BuildValue if unlikely $tmp is pos;
$rd_exp_val = $rd_exp_val * 10;
$tmp = $c - '0';
$rd_exp_val = $rd_exp_val + $tmp;
$c = [$line_ptr].bu;
```

```
  $line_ptr = $line_ptr + 1;
  goto ExponentDigits;

  BuildValue:
  $value_f = flot. $rd_int_part;
  $tmp = $rd_frac_scale - 1;
  goto ApplyExponent if $tmp is zero;   # frac_scale == 1: no
   fractional digits seen
  $frac_f = flot. $rd_frac_part;
  $scale_f = flot. $rd_frac_scale;
  $frac_f = $frac_f /. $scale_f;
  $value_f = $value_f +. $frac_f;

  ApplyExponent:
  goto ApplySign if $rd_exp_val is zero;

  ExponentLoop:
  goto DivideByTen if $rd_exp_sign is nonzero;
  $value_f = $value_f *. $ten_f;
  goto ExponentStepDone;

  DivideByTen:
  $value_f = $value_f /. $ten_f;

  ExponentStepDone:
  $rd_exp_val = $rd_exp_val - 1;
  goto ExponentLoop if likely $rd_exp_val is nonzero;

  ApplySign:
  goto ReturnValue if $rd_sign is zero;
  $value_f = $zero_f -. $value_f;

  ReturnValue:
  return $value_f;

  EndRead:
  return $zero_f;
end

@* The output formatter.

\texttt{FormatLine} builds a complete output line in \texttt{OutBuf}
   and
returns its address for \verb|FPUTS|.  Two cases:

\begin{itemize}
\item If $y>400$, write the index digits and a space, then copy
      \texttt{TooLarge} (``\texttt{TOO LARGE\textbackslash n}'') byte
   by byte.
\item Otherwise, write the index, a space, an optional minus sign, and
    $y$
      in \texttt{\%.16g} format: 16 significant digits, fixed notation
    when
      the decimal exponent lies in $[-4,\,15]$, scientific notation
   otherwise.
\end{itemize}

\medskip
```

```
\textbf{Buffer layout.}
\texttt{OutBuf} is 96 bytes wide and is used in two zones during
   formatting.
The \emph{output zone} (bytes 0--\texttt{DIGIT\_START}$-1$)
   accumulates the
characters that will eventually be printed: index, sign, digits,
   decimal
point, and exponent suffix.  The \emph{scratch zone}
(bytes \texttt{DIGIT\_START}--\texttt{DIGIT\_LAST}) is a fixed 16-byte
    area
where the mantissa digits are written right-to-left during extraction,
    before
being copied forward into the output zone.  The two zones never
   overlap
because the output prefix is at most a few characters wide.

\medskip
\textbf{Normalisation.}
The absolute value is scaled by repeated division or multiplication by
    10.0
until it lies in $[1,10)$, tracking the decimal exponent \verb|$exp10
   |.
The normalised value is then multiplied by $10^{15}$ (\texttt{NUM\
   _DIGITS}$-1$
steps, since the value already provides the leading digit) and rounded
    to the
nearest integer to obtain a 16-digit integer mantissa.  If rounding
   pushes the
mantissa to $10^{16}$ (\texttt{MANT\_OVERFLOW}), it is reset to $10
   ^{15}$
(\texttt{MANT\_UNIT}) and \verb|$exp10| is incremented.

\medskip
\textbf{Notation choice.}  After the mantissa digits are extracted and
trailing zeros and a possible trailing decimal point are trimmed,
\texttt{FinishNumericLine} appends an exponent suffix when
$\mathtt{exp10}<-4$ or $\mathtt{exp10}\ge16$; otherwise the number
   already
appears in fixed form.

\medskip
\textbf{Control flow.}
After extracting the 16 mantissa digits into the scratch zone,
\texttt{FormatLine} branches on \verb|$exp10| to one of three
   formatting
paths: \texttt{FixedIntegerPart} (when $0\le\mathtt{exp10}\le15$),
\texttt{FixedLeadingZero} (when $-4\le\mathtt{exp10}<0$), and
\texttt{ScientificForm} (when $\mathtt{exp10}<-4$ or $\mathtt{exp10}\
   ge16$).
All three copy digits from the scratch zone into the output zone and
   then
merge at \texttt{TrimDecimal}, which strips trailing zeros and a
   possible
lone decimal point.  Control then falls to \texttt{FinishNumericLine},
    which
appends the exponent suffix if required, and finally to \texttt{
   DoneLine},
which appends the newline and null terminator.
```

```
\medskip
\textbf{The \texttt{TooLarge} string.}
The error message belongs logically with the output formatter, so its
definition appears here.  Its physical position in the assembled data
segment is controlled by the chunk reference in section~3: immediately
after \texttt{InputArray}, it fills the gap needed to align the first
\verb|greg| that follows.

@<TooLarge data@>=
byte TooLarge "TOO LARGE\n", 0;
reserve 5;   # InputArray: 11*8 = 88 bytes; TooLarge string: 11 bytes;
    total 99; +5 = 104 = 8*13

@<Data segment@>=
greg $OutBuf[96];

@<Output helper@>=
def FormatLine($index, $value) -> $line:
  # OutBuf layout (see section 6 prose):
  #   output zone:  bytes 0 .. DIGIT_START-1
  #   scratch zone: bytes DIGIT_START .. DIGIT_LAST  (16 digits, right
   -to-left)
  const DIGIT_START    = 32;
  const DIGIT_LAST     = 47;
  const DIGIT_STOP     = 48;
  const NUM_DIGITS     = 16;
  const MANT_OVERFLOW  = 10000000000000000;   # 10^16: rounding
   carried into next power
  const MANT_UNIT      = 1000000000000000;    # 10^15: reset value on
   overflow
  register $pos, $tmp, $src, $ch, $abs_f, $norm_f, $scaled_f, $cmp_f,
   $mant;
  register $exp10, $digits_left, $exp_abs, $trim_pos, $digit_ptr,
   $digit;
  register $copy_idx, $zeros;

  $pos = 0;

  $tmp = $index - 10;
  goto OneDigitIndex if likely $tmp is neg;
  $OutBuf[$pos].bu = '1';
  $pos = $pos + 1;
  $index = $tmp;

  OneDigitIndex:
  $digit = $index + '0';
  $OutBuf[$pos].bu = $digit;
  $pos = $pos + 1;
  $OutBuf[$pos].bu = ' ';
  $pos = $pos + 1;

  $cmp_f = $value -. $threshold_f;
  goto FormatNumeric if likely $cmp_f is nonpos;

  $src = &TooLarge;
  CopyTooLarge:
  $ch = [$src].bu;
```

```
$OutBuf[$pos].bu = $ch;
$src = $src + 1;
$pos = $pos + 1;
goto CopyTooLarge if likely $ch is nonzero;
return $OutBuf;

FormatNumeric:
$cmp_f = $value -. $zero_f;
goto MakeAbsolute if likely $cmp_f is nonneg;
$OutBuf[$pos].bu = '-';
$pos = $pos + 1;

MakeAbsolute:
$abs_f = $value;
$abs_f andnh= #8000;
goto FormatNonzero if likely $abs_f is nonzero;
$OutBuf[$pos].bu = '0';
$pos = $pos + 1;
goto DoneLine;

FormatNonzero:
$norm_f = $abs_f;
$exp10 = 0;

NormalizeHigh:
$cmp_f = $norm_f -. $ten_f;
goto NormalizeLow if $cmp_f is neg;
$norm_f = $norm_f /. $ten_f;
$exp10 = $exp10 + 1;
goto NormalizeHigh;

NormalizeLow:
$cmp_f = $norm_f -. $one_f;
goto BuildMantissa if $cmp_f is nonneg;
$norm_f = $norm_f *. $ten_f;
$exp10 = $exp10 - 1;
goto NormalizeLow;

BuildMantissa:
$scaled_f = $norm_f;
$digits_left = NUM_DIGITS - 1;        # norm_f in [1,10) already
 gives the leading digit

ScaleLoop:
goto RoundMantissa if unlikely $digits_left is zero;
$scaled_f = $scaled_f *. $ten_f;
$digits_left = $digits_left - 1;
goto ScaleLoop;

RoundMantissa:
$mant = fix.[nearest] $scaled_f;
$tmp = $mant - MANT_OVERFLOW;
goto MantissaReady if $tmp is nonzero;
$mant = MANT_UNIT;
$exp10 = $exp10 + 1;

MantissaReady:
$digit_ptr = DIGIT_LAST;
```

```
$digits_left = NUM_DIGITS;

MantissaDigits:
$tmp = $mant / 10;
$digit = $rR;
$digit = $digit + '0';
$OutBuf[$digit_ptr].bu = $digit;
$digit_ptr = $digit_ptr - 1;
$mant = $tmp;
$digits_left = $digits_left - 1;
goto MantissaDigits if likely $digits_left is nonzero;

$tmp = $exp10 + 4;
goto ScientificForm if $tmp is neg;
$tmp = $exp10 - 16;
goto ScientificForm if $tmp is nonneg;

goto FixedLeadingZero if $exp10 is neg;

FixedIntegerPart:
$copy_idx = DIGIT_START;
$digits_left = $exp10 + 1;

CopyIntegerDigits:
$digit = $OutBuf[$copy_idx].bu;
$OutBuf[$pos].bu = $digit;
$pos = $pos + 1;
$copy_idx = $copy_idx + 1;
$digits_left = $digits_left - 1;
goto CopyIntegerDigits if likely $digits_left is nonzero;
$tmp = $exp10 - 15;
goto FinishNumericLine if $tmp is zero;
$OutBuf[$pos].bu = '.';
$pos = $pos + 1;

CopyFixedFraction:
$tmp = $copy_idx - DIGIT_STOP;
goto TrimDecimal if unlikely $tmp is zero;
$digit = $OutBuf[$copy_idx].bu;
$OutBuf[$pos].bu = $digit;
$pos = $pos + 1;
$copy_idx = $copy_idx + 1;
goto CopyFixedFraction;

FixedLeadingZero:
$OutBuf[$pos].bu = '0';
$pos = $pos + 1;
$OutBuf[$pos].bu = '.';
$pos = $pos + 1;
$zeros = - $exp10;
$zeros = $zeros - 1;

CopyFixedZeros:
goto CopyAllDigits if $zeros is zero;
$OutBuf[$pos].bu = '0';
$pos = $pos + 1;
$zeros = $zeros - 1;
goto CopyFixedZeros;
```

```
CopyAllDigits:
$copy_idx = DIGIT_START;

CopyAllDigitsLoop:
$tmp = $copy_idx - DIGIT_STOP;
goto TrimDecimal if unlikely $tmp is zero;
$digit = $OutBuf[$copy_idx].bu;
$OutBuf[$pos].bu = $digit;
$pos = $pos + 1;
$copy_idx = $copy_idx + 1;
goto CopyAllDigitsLoop;

ScientificForm:
$digit = $OutBuf[DIGIT_START].bu;
$OutBuf[$pos].bu = $digit;
$pos = $pos + 1;
$OutBuf[$pos].bu = '.';
$pos = $pos + 1;
$copy_idx = DIGIT_START + 1;

CopyScientificDigits:
$tmp = $copy_idx - DIGIT_STOP;
goto TrimDecimal if unlikely $tmp is zero;
$digit = $OutBuf[$copy_idx].bu;
$OutBuf[$pos].bu = $digit;
$pos = $pos + 1;
$copy_idx = $copy_idx + 1;
goto CopyScientificDigits;

TrimDecimal:
$trim_pos = $pos - 1;

TrimTrailingZeros:
$digit = $OutBuf[$trim_pos].bu;
$tmp = $digit - '0';
goto TrimPoint if $tmp is nonzero;
$pos = $pos - 1;
$trim_pos = $trim_pos - 1;
goto TrimTrailingZeros;

TrimPoint:
$digit = $OutBuf[$trim_pos].bu;
$tmp = $digit - '.';
goto FinishNumericLine if $tmp is nonzero;
$pos = $pos - 1;

FinishNumericLine:
$tmp = $exp10 + 4;
goto DoneLine if $tmp is nonneg;
$tmp = $exp10 - 16;
goto DoneLine if $tmp is neg;
$OutBuf[$pos].bu = 'e';
$pos = $pos + 1;
goto PositiveExponent if $exp10 is nonneg;
$OutBuf[$pos].bu = '-';
$pos = $pos + 1;
$exp_abs = - $exp10;
```

```
  goto WriteExponentDigits;

  PositiveExponent:
  $OutBuf[$pos].bu = '+';
  $pos = $pos + 1;
  $exp_abs = $exp10;

  WriteExponentDigits:
  $tmp = $exp_abs / 100;
  goto ExponentTens if likely $tmp is zero;
  $digit = $tmp + '0';
  $OutBuf[$pos].bu = $digit;
  $pos = $pos + 1;
  $exp_abs = $rR;

  ExponentTens:
  $tmp = $exp_abs / 10;
  $digit = $tmp + '0';
  $OutBuf[$pos].bu = $digit;
  $pos = $pos + 1;
  $digit = $rR;
  $digit = $digit + '0';
  $OutBuf[$pos].bu = $digit;
  $pos = $pos + 1;

  DoneLine:
  $OutBuf[$pos].bu = '\n';
  $pos = $pos + 1;
  $OutBuf[$pos].bu = '\0';
  return $OutBuf;
end

@* Build and weave.

To build this program:

\begin{verbatim}
$ plmmix-tangle -o tpk.plm tpk.plw
$ plmmix-compiler tpk.plm -o tpk.mms
$ mmixal -o tpk.o tpk.mms
$ mmix tpk.o
\end{verbatim}

To create a typeset version with cross-references:

\begin{verbatim}
$ plmmix-weave -o tpk.tex tpk.plw
$ pdflatex tpk.tex
\end{verbatim}
```

A Grammar Summary

"For last year's words belong to last year's language
and next year's words await another voice."
— T. S. Eliot, *Little Gidding* (1942)

This section collects the complete grammar of PL/MMIX for quick reference.

Top-level structure and declarations

```
Program ::= { ProcDecl | DataDecl | SegmentDirective | LocDirective |
ReserveDirective
              | RegisterDecl | ConstDecl | PrefixDirective | LocalDecl |
IncludeDirective | PoolDirective
              | Statement }

ProcDecl ::= def Label "(" ProcParams ")" [ "->" ProcParams ] ":" StatementList end
ProcParams ::= [ ProcParam { "," ProcParam } ]
ProcParam ::= Register | ParamName "=" Register | ParamName
ParamName ::= Label | RegName
StatementList ::= { LabelDecl | Statement }
LabelDecl ::= Label ":"

DataDecl ::= greg GregName ( "[" ConstExpr "]" | [ "=" GregInit ] ) ";"
| byte Label DataLiteral { "," DataLiteral } ";"
| wyde Label DataLiteral { "," DataLiteral } ";"
| tetra Label DataLiteral { "," DataLiteral } ";"
| octa Label DataLiteral { "," DataLiteral } ";"
DataLiteral ::= [ "-" ] Immediate | Label | StringLit   (Immediate includes
CharLit)
GregName ::= Label | RegName
GregInit ::= Label [ ("+" | "-") ConstExpr ]
| ConstExpr
ConstExpr ::= ConstTerm { ("+" | "-") ConstTerm }
ConstTerm ::= ConstAtom { ("*" | "/") ConstAtom }
ConstAtom ::= [ "-" ] Immediate | "(" ConstExpr ")"
SegmentDirective ::= segment (data | code) [ GregName ] ";"
PoolDirective ::= pool ";"
LocDirective ::= loc ConstExpr ";"
ReserveDirective ::= reserve [ Label ] ConstExpr ";"
RegisterDecl ::= register RegName "=" Register ";"
| register RegName ";"
| register RegName { "," RegName } ";"
ConstDecl ::= const Label "=" ConstExpr ";"
ConstExpr ::= ConstAtom { ("+" | "-" | "*" | "/") ConstAtom } | "(" ConstExpr ")"
ConstAtom ::= [ "-" ] Immediate | ConstAliasName
PrefixDirective ::= prefix [ Label ] ";"
LocalDecl ::= local Register ";"   (explicit only; MMIXAL infers local register
count automatically)
IncludeDirective ::= include StringLit ";"

BeginBlock ::= begin ":" StatementList end

Statement ::= Assignment | CondSetStatement | ZeroSetStatement
| LoadStatement | StoreStatement | GetaStatement
| SpecialRegAssign | RoundStmt | WydeSetStatement
| ConstDecl | RegisterDecl
| BeginBlock
| IfStatement | PostfixIfStatement | WhileStatement | DoWhileStatement |
LoopStatement
```

```
| BreakStatement | ContinueStatement | GotoStatement
| CallStatement | ReturnStatement
| CacheStatement | SyncStatement | CswapStatement
| SaveStatement | LdvtsStatement
| TrapStatement | TripStatement | ResumeStatement
```

Expressions and assignments

```
RegOperand ::= Register | RegName   (register alias; see §4)
Assignment ::= RegOperand "=" Expression ";"
Expression ::= UnaryOp RegOperand | RegOperand BinaryOp RegOperand
| RegOperand BinaryOp SmallImm | RegOperand | Immediate

CondSetStatement ::= RegOperand "=" CondVal if RegOperand is Polarity ";"
ZeroSetStatement ::= RegOperand "?=" CondVal if RegOperand is Polarity ";"
CondVal ::= RegOperand | SmallImm
Polarity ::= neg|zero|pos|odd|nonneg|nonzero|nonpos|even
```

Memory, address-of, wyde-immediate, and special registers

```
LoadStatement ::= RegOperand "=" "[" Address "]" SizeSuffix ";"
| RegOperand "=" "[" Address "]" ".unc" ";"
StoreStatement ::= "[" Address "]" SizeSuffix "=" StoreRhs ";"
| "[" Address "]" ".unc" "=" RegOperand ";"
| "[" Address "]" ".o" "=" StoreImm ";"
StoreRhs ::= RegOperand | StoreImm
StoreImm ::= SmallImm | ConstAliasName | CharLiteral
Address ::= RegOperand "+" RegOperand | RegOperand "+" SmallImm | RegOperand
SizeSuffix ::= ".b"|".bu"|".w"|".wu"|".t"|".tu"|".o"|".sf"|".ht"

GetaStatement ::= RegOperand "=" "&" Label ";"   (GETA for code-segment labels;
auto-GREG+SET for data-segment labels)

WydeSetStatement ::= RegOperand WydeOp WydeImm ";"
WydeOp ::= orh=|ormh=|orml=|orl=|andnh=|andnmh=|andnml=|andnl=

SpecialRegAssign ::= UserSpecialReg "=" RegOperand ";"
| PrivSpecialReg "=" RegOperand ";"
| RegOperand "=" AnySpecialReg ";"
RoundStmt ::= rounding RoundMode ";"
RoundMode ::= nearest|zero|up|down
```

Control flow and procedure calls

```
IfStatement ::= if [BranchHint] Condition ":" StatementList
{ elif [BranchHint] Condition ":" StatementList }
[else ":" StatementList] end
PostfixIfStatement ::= SimpleStatement if [BranchHint] Condition ";"
WhileStatement ::= while [BranchHint] Condition ":" StatementList end
DoWhileStatement ::= do ":" StatementList end while [BranchHint] Condition ";"
LoopStatement ::= loop ":" StatementList end
BreakStatement ::= break ";"
ContinueStatement ::= continue ";"
BranchHint ::= likely | unlikely
Condition ::= RegOperand CompareOp RegOperand | RegOperand CompareOp SmallImm
| RegOperand FloatCompareOp RegOperand | RegOperand is Polarity
GotoStatement ::= goto Label ";" | goto "[" RegOperand "]" ";"
```

```
CallStatement ::= [RegList "="] call Label "(" ArgList ")" ";"
| [RegList "="] call "[" RegOperand "]" "(" ArgList ")" ";"
ReturnStatement ::= return [ReturnExpr {"," ReturnExpr}] ";"
ReturnExpr ::= RegOperand | ["-"] Immediate | CharLit | Label | call Label "("
ArgList ")"
RegList ::= RegOperand { "," RegOperand }
ArgList ::= [ ArgItem { "," ArgItem } ];   ArgItem ::= single assignment atom
```

Cache, memory order, and system statements

```
CacheStatement ::= (preld|prego|prest|syncd|syncid) SmallImm "," Address ";"
SyncStatement ::= sync SmallImm ";"
CswapStatement ::= RegOperand "=" cswap "[" Address "]" ";"
SaveStatement ::= (save|unsave) RegOperand ";"
LdvtsStatement ::= ldvts RegOperand "," Address ";"
TrapStatement ::= trap SmallImm "," SmallImm "," SmallImm ";"
TripStatement ::= trip SmallImm "," SmallImm "," SmallImm ";"
ResumeStatement::= resume [SmallImm] ";"
```

Terminals

```
Register ::= "$" DecDigit { DecDigit }   (0-255)
SmallImm ::= DecDigit{DecDigit} | "#" HexDigit HexDigit   (0-255)
WydeImm ::= DecDigit{DecDigit} | "#" HexDigit^4   (0-65535)
Immediate ::= DecDigit{DecDigit} | "#" HexDigit{HexDigit} | CharLit
CharLit ::= "'" ( Char | EscapeSeq ) "'"   (Unicode ordinal of character)
Label ::= Letter { Letter | DecDigit | "_" }
```

Reserved words

See Section 2.5 for the complete inventory of reserved identifiers and operator lexemes.

B TPK Algorithm: Woven Output

"Long is the road through precepts; short and effective through examples."
— Seneca, *Epistulae Morales*, VI.5 (c. 65 AD)

This appendix shows the complete output of running `plmmix-weave` on `examples/tpk.plw` (the source is shown in Section 16.8). It demonstrates how the literate programming tools transform the `.plw` source into a typeset document with hyperlinked code chunks, chunk definitions and usage references, and a comprehensive cross-reference index.

1. The TPK Algorithm in PL/MMIX.

The TPK algorithm (Trabb Pardo–Knuth, 1977) is a small benchmark program designed to exercise a language's essential features: input, output, arrays, functions, and floating-point arithmetic. It reads $N = 11$ values $t_0, \ldots, t_{10}$, then for each index i from 10 down to 0 computes

$$y = \sqrt{\texttt{t_i}} + 5t_i^3$$

and prints either "i TOO LARGE" if $y > 400$, or "i y" with y formatted to 16 significant digits.

This PL/MMIX implementation handles two non-trivial helper procedures explicitly: decimal floating-point input is parsed by `ReadNumber`, and the result is formatted by `FormatLine`; the main loop simply calls them. Supporting data is interleaved with the code in each section rather than gathered in one place, keeping each datum close to the code that uses it.

2. Program structure.

The MMIX convention places the data segment at address `#2000000000000000` and the code segment at `#100`. All data declarations accumulate in `@<Data segment@>`, which therefore appears first in the assembled file.

In the code segment the two helper procedures reside after the `HALT` instruction. The main body executes first and falls into `HALT`; the helpers are reached only via `PUSHJ`, which can reach addresses both before and after the call site (within ±256 K instructions), so placing helpers after `HALT` is entirely valid. This keeps the main body as a single unbroken sequence of instructions with no helper code interspersed.

Register `$255` is MMIX's conventional trap I/O channel: `FGETS` deposits its result count there, and `FPUTS` reads the string address from it. The alias `$out` is used throughout so call sites remain readable.

⟨ ***Program*** ⟩ ≡

```
⟨ Data segment ⟩
 segment  code;
 register $out = $255;
⟨ Read input values ⟩
⟨ Process in reverse ⟩
 trap 0, HALT, 0;
⟨ Input helpers ⟩
⟨ Output helper ⟩
```

3. Reading the inputs.

The input loop calls `ReadNumber` N times, storing each result into successive octabytes of `InputArray`. Register `$addr` walks the array; `$count` counts down so the loop needs no separate index.

⟨ ***Data segment*** ⟩ ≡

```
const N = 11;                              # number of input values
segment  data $DataBase;
octa InputArray 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
⟨TooLarge data⟩
greg $count = 0;
greg $addr = 0;
greg $t = 0;
```

⟨ ***Read input values*** ⟩ ≡

```
$addr = &InputArray;
$count = N;
while  likely $count != 0:
 $t =  call ReadNumber();
 [$addr].o = $t;
 $addr = $addr + 8;
 $count = $count - 1;
end
```

4. Computing and printing in reverse.

After the read pass `$addr` points one element past the array, so the reverse walk steps it back by one octabyte first.

The absolute value of t is obtained by clearing the IEEE 754 sign bit (`andnh= #8000`) before passing the value to `sqrt.`. The cubic term is computed as $t \cdot t \cdot t \cdot 5$, accumulating into the same register without a temporary.

The loop's three work registers are `$i` (current index), `$u` (absolute value of t, computed in-place before the square root), and `$y` (accumulated result). Of the five floating-point constants, `$five_f` and `$threshold_f` are specific to this algorithm; `$zero_f`, `$one_f`, and `$ten_f` are used mainly by the helper procedures and are declared here because all global registers share a single GREG pool. All five are given literal float initialisers: PL/MMIX folds each to its IEEE 754 bit pattern at compile time, so no runtime `FLOT` instruction is needed.

⟨ ***Data segment*** ⟩ +≡

```
greg $i = 0;
greg $u = 0;
greg $y = 0;
greg $five_f = 5.0;
greg $threshold_f = 400.0;
greg $zero_f = 0.0;
greg $one_f = 1.0;
greg $ten_f = 10.0;
```

⟨ ***Process in reverse*** ⟩ ≡

```
$addr = $addr - 8;
$i = N - 1;
$count = N;

 while  likely $count != 0:
  $t = [$addr].o;

  $u = $t;
  $u andnh= #8000;              # |t|: clear sign bit
  $u = sqrt. $u;

  $y = $t *. $t;
  $y = $y *. $t;                # t^3
  $y = $y *. $five_f;           # 5t^3
  $y = $u +. $y;                # sqrt(|t|) + 5t^3

  $out =  call FormatLine($i, $y);
   trap 0, FPUTS, STDOUT;

  $addr = $addr - 8;
  $i = $i - 1;
  $count = $count - 1;
 end
```

5. The floating-point input parser.

`ReadNumber` reads one line from standard input via `FGETS` and parses the first decimal floating-point token on that line. It recognises an optional sign, an integer part, an optional fractional part, and an optional decimal exponent (`e` or `E`). Before reading the number, any leading bytes with value at most 32 are skipped—this covers spaces, tabs, carriage returns, and newlines alike.

The value is assembled in three stages. First the integer and fractional parts are accumulated as integers (`$rd_int_part` and `$rd_frac_part`), with `$rd_frac_scale` counting the implicit denominator (initially 1, multiplied by 10 for each fractional digit). Then both are converted to float and combined. Finally the exponent loop rescales by repeated multiplication or division by 10.0.

The six mutable parser state registers are globals; they are reset at the top of every call. The line buffer `$InputBuf` holds up to 127 characters plus a null terminator; `$line\_ptr` advances byte by byte through the buffer after each `FGETS`.

⟨ ***Data segment*** ⟩ +≡

```
greg $line_ptr = 0;
greg $InputBuf[128];
greg $ReadArgs;
octa ReadArgsData 0, 127;
greg $rd_sign = 0;
greg $rd_int_part = 0;
greg $rd_frac_part = 0;
greg $rd_frac_scale = 1;
greg $rd_exp_sign = 0;
greg $rd_exp_val = 0;
```

⟨ ***Input helpers*** ⟩ ≡

```
⟨Read one floating-point value procedure⟩
```

⟨ ***Read one floating-point value procedure*** ⟩ ≡

```
def ReadNumber() -> $result:
  register $c = $0;
  register $tmp, $value_f, $frac_f, $scale_f;


 [$ReadArgs].o = $InputBuf;
 $out = $ReadArgs;
  trap 0, FGETS, STDIN;
  goto EndRead  if  unlikely $out  is  nonpos;
 $line_ptr = $InputBuf;

 $rd_sign = 0;
 $rd_int_part = 0;
 $rd_frac_part = 0;
 $rd_frac_scale = 1;
 $rd_exp_sign = 0;
 $rd_exp_val = 0;

 SkipSpace:
 $c = [$line_ptr].bu;
  goto EndRead  if  unlikely $c  is  zero;
 $line_ptr = $line_ptr + 1;
 $tmp = $c  cmpu ' ';
  goto SkipSpace  if $tmp  is  nonpos;

 $tmp = $c  cmpu '-';
  goto CheckPlus  if  likely $tmp  is  nonzero;
 $rd_sign = 1;
 $c = [$line_ptr].bu;
 $line_ptr = $line_ptr + 1;
  goto IntegerDigits;

 CheckPlus:
 $tmp = $c  cmpu '+';
  goto IntegerDigits  if  likely $tmp  is  nonzero;
 $c = [$line_ptr].bu;
 $line_ptr = $line_ptr + 1;

 IntegerDigits:
```

```
$tmp = $c  cmpu '0';
 goto AfterInteger  if  unlikely $tmp  is  neg;
$tmp = $c  cmpu '9';
 goto AfterInteger  if  unlikely $tmp  is  pos;
$rd_int_part = $rd_int_part * 10;
$tmp = $c - '0';
$rd_int_part = $rd_int_part + $tmp;
$c = [$line_ptr].bu;
$line_ptr = $line_ptr + 1;
 goto IntegerDigits;

AfterInteger:
$tmp = $c  cmpu '.';
 goto AfterFraction  if $tmp  is  nonzero;
$c = [$line_ptr].bu;
$line_ptr = $line_ptr + 1;

FractionDigits:
$tmp = $c  cmpu '0';
 goto AfterFraction  if  unlikely $tmp  is  neg;
$tmp = $c  cmpu '9';
 goto AfterFraction  if  unlikely $tmp  is  pos;
$rd_frac_part = $rd_frac_part * 10;
$tmp = $c - '0';
$rd_frac_part = $rd_frac_part + $tmp;
$rd_frac_scale = $rd_frac_scale * 10;
$c = [$line_ptr].bu;
$line_ptr = $line_ptr + 1;
 goto FractionDigits;

AfterFraction:
$tmp = $c  cmpu 'E';
 goto HaveExponent  if $tmp  is  zero;
$tmp = $c  cmpu 'e';
 goto HaveExponent  if $tmp  is  zero;
 goto BuildValue;

HaveExponent:
$c = [$line_ptr].bu;
$line_ptr = $line_ptr + 1;
$tmp = $c  cmpu '-';
 goto CheckExponentPlus  if $tmp  is  nonzero;
$rd_exp_sign = 1;
$c = [$line_ptr].bu;
$line_ptr = $line_ptr + 1;
 goto ExponentDigits;

CheckExponentPlus:
$tmp = $c  cmpu '+';
 goto ExponentDigits  if $tmp  is  nonzero;
$c = [$line_ptr].bu;
$line_ptr = $line_ptr + 1;

ExponentDigits:
$tmp = $c  cmpu '0';
 goto BuildValue  if  unlikely $tmp  is  neg;
$tmp = $c  cmpu '9';
 goto BuildValue  if  unlikely $tmp  is  pos;
$rd_exp_val = $rd_exp_val * 10;
$tmp = $c - '0';
$rd_exp_val = $rd_exp_val + $tmp;
$c = [$line_ptr].bu;
```

```
    $line_ptr = $line_ptr + 1;
     goto ExponentDigits;

    BuildValue:
    $value_f = flot. $rd_int_part;
    $tmp = $rd_frac_scale - 1;
     goto ApplyExponent  if $tmp  is  zero;    # frac_scale == 1: no fractional digits seen
    $frac_f = flot. $rd_frac_part;
    $scale_f = flot. $rd_frac_scale;
    $frac_f = $frac_f /. $scale_f;
    $value_f = $value_f +. $frac_f;

    ApplyExponent:
     goto ApplySign  if $rd_exp_val  is  zero;

    ExponentLoop:
     goto DivideByTen  if $rd_exp_sign  is  nonzero;
    $value_f = $value_f *. $ten_f;
     goto ExponentStepDone;

    DivideByTen:
    $value_f = $value_f /. $ten_f;

    ExponentStepDone:
    $rd_exp_val = $rd_exp_val - 1;
     goto ExponentLoop  if  likely $rd_exp_val  is  nonzero;

    ApplySign:
     goto ReturnValue  if $rd_sign  is  zero;
    $value_f = $zero_f -. $value_f;

    ReturnValue:
     return $value_f;

    EndRead:
     return $zero_f;
   end
```

6. The output formatter.

`FormatLine` builds a complete output line in `OutBuf` and returns its address for `FPUTS`. Two cases:

- If $y > 400$, write the index digits and a space, then copy `TooLarge` ("`TOO LARGE\n`") byte by byte.
- Otherwise, write the index, a space, an optional minus sign, and y in `%.16g` format: 16 significant digits, fixed notation when the decimal exponent lies in $[-4, 15]$, scientific notation otherwise.

Buffer layout. `OutBuf` is 96 bytes wide and is used in two zones during formatting. The *output zone* (bytes 0–`DIGIT_START`-1) accumulates the characters that will eventually be printed: index, sign, digits, decimal point, and exponent suffix. The *scratch zone* (bytes `DIGIT_START`–`DIGIT_LAST`) is a fixed 16-byte area where the mantissa digits are written right-to-left during extraction, before being copied forward into the output zone. The two zones never overlap because the output prefix is at most a few characters wide.

Normalisation. The absolute value is scaled by repeated division or multiplication by 10.0 until it lies in $[1, 10)$, tracking the decimal exponent `$exp10`. The normalised value is then multiplied by 10^{15} (`NUM_DIGITS`-1 steps, since the value already provides the leading digit) and rounded to the nearest integer to obtain a 16-digit integer mantissa. If rounding pushes the mantissa to 10^{16} (`MANT_OVERFLOW`), it is reset to 10^{15} (`MANT_UNIT`) and `$exp10` is incremented.

Notation choice. After the mantissa digits are extracted and trailing zeros and a possible trailing decimal point are trimmed, `FinishNumericLine` appends an exponent suffix when `exp10` < -4 or `exp10` ≥ 16; otherwise the number already appears in fixed form.

Control flow. After extracting the 16 mantissa digits into the scratch zone, `FormatLine` branches on `$exp10` to one of three formatting paths: `FixedIntegerPart` (when $0 \leq$ `exp10` ≤ 15), `FixedLeadingZero` (when $-4 \leq$ `exp10` < 0), and `ScientificForm` (when `exp10` < -4 or `exp10` ≥ 16). All three copy digits from the scratch zone into the output zone and then merge at `TrimDecimal`, which strips trailing zeros and a possible lone decimal point. Control then falls to `FinishNumericLine`, which appends the exponent suffix if required, and finally to `DoneLine`, which appends the newline and null terminator.

The `TooLarge` string. The error message belongs logically with the output formatter, so its definition appears here. Its physical position in the assembled data segment is controlled by the chunk reference in section 3: immediately after `InputArray`, it fills the gap needed to align the first `greg` that follows.

⟨ ***TooLarge data*** ⟩ ≡

```
byte TooLarge "TOO LARGE\n", 0;
reserve 5;    # InputArray: 11*8 = 88 bytes; TooLarge string: 11 bytes; total 99; +5 = 104 =
 8*13
```

⟨ ***Data segment*** ⟩ +≡

```
greg $OutBuf[96];
```

⟨ ***Output helper*** ⟩ ≡

```
def FormatLine($index, $value) -> $line:
  # OutBuf layout (see section 6 prose):
  #   output zone:  bytes 0 .. DIGIT_START-1
  #   scratch zone: bytes DIGIT_START .. DIGIT_LAST  (16 digits, right-to-left)


  const DIGIT_START    = 32;
  const DIGIT_LAST     = 47;
  const DIGIT_STOP     = 48;
  const NUM_DIGITS     = 16;
  const MANT_OVERFLOW  = 10000000000000000;    # 10^16: rounding carried into next power
  const MANT_UNIT      = 1000000000000000;     # 10^15: reset value on overflow
  register $pos, $tmp, $src, $ch, $abs_f, $norm_f, $scaled_f, $cmp_f, $mant;
  register $exp10, $digits_left, $exp_abs, $trim_pos, $digit_ptr, $digit;
  register $copy_idx, $zeros;

 $pos = 0;

 $tmp = $index - 10;
```

```
 goto OneDigitIndex  if  likely $tmp  is  neg;
$OutBuf[$pos].bu = '1';
$pos = $pos + 1;
$index = $tmp;

OneDigitIndex:
$digit = $index + '0';
$OutBuf[$pos].bu = $digit;
$pos = $pos + 1;
$OutBuf[$pos].bu = ' ';
$pos = $pos + 1;

$cmp_f = $value -. $threshold_f;
 goto FormatNumeric  if  likely $cmp_f  is  nonpos;

$src = &TooLarge;
CopyTooLarge:
$ch = [$src].bu;
$OutBuf[$pos].bu = $ch;
$src = $src + 1;
$pos = $pos + 1;
 goto CopyTooLarge  if  likely $ch  is  nonzero;
 return $OutBuf;

FormatNumeric:
$cmp_f = $value -. $zero_f;
 goto MakeAbsolute  if  likely $cmp_f  is  nonneg;
$OutBuf[$pos].bu = '-';
$pos = $pos + 1;

MakeAbsolute:
$abs_f = $value;
$abs_f andnh= #8000;
 goto FormatNonzero  if  likely $abs_f  is  nonzero;
$OutBuf[$pos].bu = '0';
$pos = $pos + 1;
 goto DoneLine;

FormatNonzero:
$norm_f = $abs_f;
$exp10 = 0;

NormalizeHigh:
$cmp_f = $norm_f -. $ten_f;
 goto NormalizeLow  if $cmp_f  is  neg;
$norm_f = $norm_f /. $ten_f;
$exp10 = $exp10 + 1;
 goto NormalizeHigh;

NormalizeLow:
$cmp_f = $norm_f -. $one_f;
 goto BuildMantissa  if $cmp_f  is  nonneg;
$norm_f = $norm_f *. $ten_f;
$exp10 = $exp10 - 1;
 goto NormalizeLow;

BuildMantissa:
$scaled_f = $norm_f;
$digits_left = NUM_DIGITS - 1;          # norm_f in [1,10) already gives the leading digit

ScaleLoop:
 goto RoundMantissa  if  unlikely $digits_left  is  zero;
```

```
$scaled_f = $scaled_f *. $ten_f;
$digits_left = $digits_left - 1;
 goto ScaleLoop;

RoundMantissa:
$mant = fix.[ nearest] $scaled_f;
$tmp = $mant - MANT_OVERFLOW;
 goto MantissaReady  if $tmp  is  nonzero;
$mant = MANT_UNIT;
$exp10 = $exp10 + 1;

MantissaReady:
$digit_ptr = DIGIT_LAST;
$digits_left = NUM_DIGITS;

MantissaDigits:
$tmp = $mant / 10;
$digit = $rR;
$digit = $digit + '0';
$OutBuf[$digit_ptr].bu = $digit;
$digit_ptr = $digit_ptr - 1;
$mant = $tmp;
$digits_left = $digits_left - 1;
 goto MantissaDigits  if  likely $digits_left  is  nonzero;

$tmp = $exp10 + 4;
 goto ScientificForm  if $tmp  is  neg;
$tmp = $exp10 - 16;
 goto ScientificForm  if $tmp  is  nonneg;

 goto FixedLeadingZero  if $exp10  is  neg;

FixedIntegerPart:
$copy_idx = DIGIT_START;
$digits_left = $exp10 + 1;

CopyIntegerDigits:
$digit = $OutBuf[$copy_idx].bu;
$OutBuf[$pos].bu = $digit;
$pos = $pos + 1;
$copy_idx = $copy_idx + 1;
$digits_left = $digits_left - 1;
 goto CopyIntegerDigits  if  likely $digits_left  is  nonzero;
$tmp = $exp10 - 15;
 goto FinishNumericLine  if $tmp  is  zero;
$OutBuf[$pos].bu = '.';
$pos = $pos + 1;

CopyFixedFraction:
$tmp = $copy_idx - DIGIT_STOP;
 goto TrimDecimal  if  unlikely $tmp  is  zero;
$digit = $OutBuf[$copy_idx].bu;
$OutBuf[$pos].bu = $digit;
$pos = $pos + 1;
$copy_idx = $copy_idx + 1;
 goto CopyFixedFraction;

FixedLeadingZero:
$OutBuf[$pos].bu = '0';
$pos = $pos + 1;
$OutBuf[$pos].bu = '.';
$pos = $pos + 1;
```

```
$zeros = - $exp10;
$zeros = $zeros - 1;

CopyFixedZeros:
 goto CopyAllDigits  if $zeros  is  zero;
$OutBuf[$pos].bu = '0';
$pos = $pos + 1;
$zeros = $zeros - 1;
 goto CopyFixedZeros;

CopyAllDigits:
$copy_idx = DIGIT_START;

CopyAllDigitsLoop:
$tmp = $copy_idx - DIGIT_STOP;
 goto TrimDecimal  if  unlikely $tmp  is  zero;
$digit = $OutBuf[$copy_idx].bu;
$OutBuf[$pos].bu = $digit;
$pos = $pos + 1;
$copy_idx = $copy_idx + 1;
 goto CopyAllDigitsLoop;

ScientificForm:
$digit = $OutBuf[DIGIT_START].bu;
$OutBuf[$pos].bu = $digit;
$pos = $pos + 1;
$OutBuf[$pos].bu = '.';
$pos = $pos + 1;
$copy_idx = DIGIT_START + 1;

CopyScientificDigits:
$tmp = $copy_idx - DIGIT_STOP;
 goto TrimDecimal  if  unlikely $tmp  is  zero;
$digit = $OutBuf[$copy_idx].bu;
$OutBuf[$pos].bu = $digit;
$pos = $pos + 1;
$copy_idx = $copy_idx + 1;
 goto CopyScientificDigits;

TrimDecimal:
$trim_pos = $pos - 1;

TrimTrailingZeros:
$digit = $OutBuf[$trim_pos].bu;
$tmp = $digit - '0';
 goto TrimPoint  if $tmp  is  nonzero;
$pos = $pos - 1;
$trim_pos = $trim_pos - 1;
 goto TrimTrailingZeros;

TrimPoint:
$digit = $OutBuf[$trim_pos].bu;
$tmp = $digit - '.';
 goto FinishNumericLine  if $tmp  is  nonzero;
$pos = $pos - 1;

FinishNumericLine:
$tmp = $exp10 + 4;
 goto DoneLine  if $tmp  is  nonneg;
$tmp = $exp10 - 16;
 goto DoneLine  if $tmp  is  neg;
$OutBuf[$pos].bu = 'e';
```

```
    $pos = $pos + 1;
     goto PositiveExponent  if $exp10  is  nonneg;
    $OutBuf[$pos].bu = '-';
    $pos = $pos + 1;
    $exp_abs = - $exp10;
     goto WriteExponentDigits;

    PositiveExponent:
    $OutBuf[$pos].bu = '+';
    $pos = $pos + 1;
    $exp_abs = $exp10;

    WriteExponentDigits:
    $tmp = $exp_abs / 100;
     goto ExponentTens  if  likely $tmp  is  zero;
    $digit = $tmp + '0';
    $OutBuf[$pos].bu = $digit;
    $pos = $pos + 1;
    $exp_abs = $rR;

    ExponentTens:
    $tmp = $exp_abs / 10;
    $digit = $tmp + '0';
    $OutBuf[$pos].bu = $digit;
    $pos = $pos + 1;
    $digit = $rR;
    $digit = $digit + '0';
    $OutBuf[$pos].bu = $digit;
    $pos = $pos + 1;

    DoneLine:
    $OutBuf[$pos].bu = '\n';
    $pos = $pos + 1;
    $OutBuf[$pos].bu = '\0';
     return $OutBuf;
   end
```

7. Build and weave.

To build this program:

```
$ plmmix-tangle -o tpk.plm tpk.plw
$ plmmix-compiler tpk.plm -o tpk.mms
$ mmixal -o tpk.o tpk.mms
$ mmix tpk.o
```

To create a typeset version with cross-references:

```
$ plmmix-weave -o tpk.tex tpk.plw
$ pdflatex tpk.tex
```

C Bootstrap Tangle: Woven Output

"Take away the dross from the silver, and there shall come forth a vessel for the finer."
— Proverbs 25:4

This appendix shows the complete woven output of the bootstrap tangler source `plmmix-tangle.plw`.

1. Introduction.

This program implements the `plmmix-tangle` step for the PL/MMIX literate programming system: it reads a `.plw` source from standard input, expands all named chunk references recursively, and writes the resulting flat PL/MMIX source to standard output.

Role in the bootstrap chain

`plmmix-tangle` is the first tool in the PL/MMIX toolchain, and it is itself written as a PL/MMIX literate program. This creates a bootstrap dependency: to build `plmmix-tangle` for the first time, an external tangle (an earlier version of this binary, or any compatible tool) must expand this file. Once built, the resulting binary can tangle any subsequent `.plw` source, including a revised version of itself.

The four build steps are:

```
plmmix-tangle -o plmmix-tangle.plm plmmix-tangle.plw
plmmix-compiler plmmix-tangle.plm -o plmmix-tangle.mms
mmixal -o plmmix-tangle.o plmmix-tangle.mms
mmix -fsource.plw plmmix-tangle.o source.plw > source.plm
```

The literate format

A `.plw` file interleaves LaTeX prose sections with PL/MMIX code chunks. Four active sigils (all beginning with an at-sign) control the structure; all other bytes are literal content. Because at-signs are active everywhere in `.plw` source, including inside LaTeX prose, they cannot be written literally in this documentation without triggering the parser. A literal at-sign is therefore written as `@@`; the examples below use that doubled-at escape. Their exact recognition rules appear in the parsing section.

`@*` Begins a major section; the heading text follows on the same line and is consumed by the parser.

`@` followed by whitespace Begins a minor prose section (no heading).

`@<`*name*`@>=` Defines (or appends to) a named chunk; the body follows until the next sigil.

`@<`*name*`@>` Inside code, expands a named chunk inline at that position.

`@@` A literal at-sign character in code or prose.

The root chunk is named `Program`; if defined, only that chunk is expanded. Otherwise all chunks are concatenated in definition order, which lets small files omit the root entirely. A chunk may be defined more than once; the bodies are appended in definition order when the chunk is expanded. Chunk names are case-sensitive; internal whitespace is normalised to a single space, so a reference and its definition may differ in spacing without causing a mismatch.

Design constraints

This is a bootstrap tool that must run on bare MMIX hardware without an operating-system allocator. The program uses no dynamic memory allocation. Instead, three fixed-size buffers are allocated at load time in the constants and data-segment sections: one for the raw input, one for normalized chunk names, and one for the assembled output. A flat fixed-size chunk table records all parsed definitions. The entire source is read in a single `FREAD` call, processed in one forward pass, and the output is flushed in a single `FWRITE` call. The output-buffer and `#line`-directive section gives the exact directive format emitted during expansion.

2. Error handling.

All errors in this program are fatal: once an anomaly is detected, the program writes a diagnostic to standard error and exits with status 1. No recovery is attempted. This is appropriate for a bootstrap tool — a partially written output file is worse than no output at all, since it might be silently passed to the next build step.

`ErrHalt` writes a null-terminated message to standard error via `FPUTS` and then halts. Every call site loads the message address into `$tmp` and passes it as the argument; callers are collected throughout the source at the exact points where the corresponding error conditions are checked.

⟨ ***Error handler*** ⟩ ≡

```
def ErrHalt($msg):
  $out = $msg;
  trap 0, FPUTS, STDERR;
  trap 0, HALT, 1;
end
```

3. Constants and limits.

All buffer sizes are fixed at compile time. If any limit is exceeded the program halts immediately with a message on standard error.

The sizes below are generous for typical PL/MMIX literate programs while remaining small enough to load quickly. They can be raised up to the limits of the MMIX address space without any other code change.

⟨ ***Constants*** ⟩ ≡

```
const SOURCE_MAX      = 524288;    # 512 KB: max source file size
const SOURCE_NAME_MAX = 1024;      # max source filename length for #line
const NAME_BUF_SIZE   = 65536;     # 64 KB:  normalized chunk-name storage
const CHUNK_MAX       = 2048;      # max number of chunk definitions
const OUT_MAX         = 1048576;   # 1 MB:   output buffer
const MAX_DEPTH       = 64;        # max expansion recursion depth (cycle guard)
const FILE_MAX        = 256;       # max number of source files after include expansion
const FILE_NAME_BUF_SIZE = 65536;  # storage for interned source filenames
const PATH_BUF_SIZE   = 2048;      # scratch for include operands and resolved paths
const TEXT_READ_MODE  = 0;         # MMIX text-read flag for FOPEN read mode
```

Each chunk table entry records where the chunk's name and body live in the fixed buffers, together with the source line of its definition marker. Five 64-bit fields are packed into 40 bytes:

Offset	Field	Meaning
0	CH_NPTR	pointer into NameBuf (normalized name start)
8	CH_NLEN	byte length of the normalized name
16	CH_BPTR	pointer into SourceBuf (raw body start)
24	CH_BLEN	byte length of the raw body
32	CH_LINE	source line number of the chunk-definition marker
40	CH_FPTR	pointer into FileNameBuf for the defining source file
48	CH_FLEN	byte length of that source-file name

The body is *not* copied: CH_BPTR and CH_BLEN describe a slice of the immutable SourceBuf. Only the name is copied (and normalized) into NameBuf, because normalization changes whitespace and the definition spelling may differ from the reference spelling.

⟨ ***Constants*** ⟩ ≡

```
const CHUNK_ENTRY_SIZE = 56;
const CHUNK_TABLE_SIZE = CHUNK_MAX * CHUNK_ENTRY_SIZE;
 # Field byte offsets within a chunk entry:
const CH_NPTR = 0;     # pointer into NameBuf  (8 bytes)
const CH_NLEN = 8;     # name length           (8 bytes)
const CH_BPTR = 16;    # pointer into SourceBuf (8 bytes)
const CH_BLEN = 24;    # body length           (8 bytes)
const CH_LINE = 32;    # source line of the chunk-definition marker
const CH_FPTR = 40;    # defining source-file pointer into FileNameBuf
const CH_FLEN = 48;    # defining source-file name length
 # Name of the root chunk (7 ASCII bytes):
const ROOT_LEN = 7;    # strlen("Program")
```

Included files need stable filename storage so `#line` directives can name the defining file of each chunk body. Each file-table entry stores a pointer and length into FileNameBuf plus a one-word state flag used to reject both duplicate inclusion and direct include cycles.

⟨ ***Constants*** ⟩ ≡

```
const FILE_ENTRY_SIZE = 24;
const FILE_TABLE_SIZE = FILE_MAX * FILE_ENTRY_SIZE;
const FILE_NPTR  = 0;
const FILE_NLEN  = 8;
const FILE_STATE = 16;
const FILE_STATE_LOADED = 1;
```

4. Data segment.

Everything lives in the data segment. The layout follows the processing pipeline: raw source enters SourceBuf, normalized chunk names accumulate in NameBuf, and assembled output grows in OutputBuf. The chunk table sits between the name and output buffers.

SourceBuf is written once by ReadAll and thereafter treated as immutable. Chunk body pointers (CH_BPTR) reference slices of this buffer directly; no body text is ever copied.

NameBuf is append-only and grows in two phases: during parsing, each chunk definition appends its normalized name; during expansion, each chunk reference also normalizes its name here for lookup. The two phases never overlap, but both use the same counter $name_end. Because

`NameBuf` only grows, the parser's name pointers (`CH_NPTR`) remain valid throughout the expansion phase.

`OutputBuf` is assembled entirely before any output is written. A single `FWRITE` call at the end of `main` flushes it to standard output.

⟨***Data segment***⟩ ≡

```
segment data $DataBase;
 # MMIX FREAD/FWRITE use two-octa argument blocks: buffer pointer, byte count.
greg $FReadArgs[16];
greg $FOpenArgs[16];


greg $FWriteArgs[16];
 # Buffers
greg $SourceBuf[SOURCE_MAX];
greg $SourceNameBuf[SOURCE_NAME_MAX];
greg $NameBuf[NAME_BUF_SIZE];
greg $FileNameBuf[FILE_NAME_BUF_SIZE];
greg $FileTable[FILE_TABLE_SIZE];
greg $ChunkTable[CHUNK_TABLE_SIZE];
greg $OutputBuf[OUT_MAX];
greg $IntBuf[32];              # decimal scratch: more than enough for 64-bit output
greg $IncludePathBuf[PATH_BUF_SIZE];
greg $ResolvedPathBuf[PATH_BUF_SIZE];
greg $IncludeStack[8 * MAX_DEPTH];
 # Global counters (all start at 0)
greg $src_len  = 0;
greg $src_name_len = 0;
greg $name_end = 0;
greg $file_name_end = 0;
greg $file_cnt = 0;
greg $chunk_cnt = 0;
greg $out_end  = 0;
greg $include_depth = 0;
greg $match_next = 0;
greg $match_path_len = 0;
greg $match_line_inc = 0;
greg $read_bptr = 0;
greg $read_blen = 0;
 # Root chunk name literal
byte RootChunk "Program", 0;
byte DefaultSourceName "stdin.plw", 0;
 # Fixed prefixes appended verbatim by the output helpers.
byte StrLineHash "#line ", 0;
```

5. Main program.

The top-level flow mirrors the three phases described in the introduction: capture the source filename from `argv[1]` for use in `#line` directives (`InitSourceName`), read the entire input into `SourceBuf` (`ReadAll`), parse it into the chunk table (`ParsePLW`), then expand the root chunk into `OutputBuf` (`ExpandRoot`) and flush it.

⟨ ***Program*** ⟩ ≡

```
⟨ Constants ⟩
⟨ Data segment ⟩
segment code;
register $out = $255;
call InitSourceName($0, $1);
call ReadAll();
call InitFileTable();
call ParsePLW($SourceBuf, $src_len, 0, $SourceNameBuf, $src_name_len);
$out_end = 0;
call ExpandRoot();
⟨ Write output to stdout ⟩
trap 0, HALT, 0;
⟨ Procedures ⟩
```

6. Procedure inventory and tangled layout.

The procedures below are declared in the order in which they appear in the tangled output: `ReadAll`, `InitSourceName`, the string utilities, the chunk-table operations, `ParsePLW`, the expansion family, the output helpers, and finally the error handler. This section is both the grep-target index for finding a routine by name and the authoritative statement of the helper order in the tangled PL/MMIX source. The narrative order that follows regroups the same procedures by topic.

⟨ ***Procedures*** ⟩ ≡

```
⟨ ReadAll procedure ⟩
⟨ InitSourceName procedure ⟩
⟨ String utilities ⟩
⟨ File and include operations ⟩
⟨ Chunk table operations ⟩
⟨ ParsePLW procedure ⟩
⟨ Expansion procedures ⟩
⟨ Output procedures ⟩
⟨ Error handler ⟩
```

7. Reading the input.

The prose now starts with `ReadAll` because bulk input handling is the first phase of the actual tangling pipeline. The auxiliary `InitSourceName` procedure still appears immediately afterward, matching its earlier call in `Program` and the tangled helper order listed in the procedure inventory.

`ReadAll` fills `SourceBuf` from standard input in one `FREAD` call, then stores the actual byte count in `$src_len`.

MMIX's `FREAD` trap takes a two-octa argument block (buffer pointer followed by requested byte count) and returns *requested_count* − *bytes_transferred* in register `$255`, so zero on a complete read. We recover the actual count as `SOURCE_MAX + return\_value`. If the source exactly fills `SOURCE_MAX` bytes we cannot tell whether more data was truncated, so we treat that case as an overflow error rather than silently producing wrong output.

⟨ ***ReadAll procedure*** ⟩ ≡

```
def ReadAll():
  register $tmp;
  $tmp = $FReadArgs;
  [$tmp].o          = $SourceBuf;
  [$tmp + 8].o      = SOURCE_MAX;
  $out = $tmp;
  trap 0, FREAD, STDIN;
  $src_len = $out + SOURCE_MAX;
  goto RASizeOk if $src_len is nonneg;
  call ErrHalt("plmmix-tangle: source too large\n");
  RASizeOk:
  $tmp = SOURCE_MAX;
  if $src_len >=u $tmp:
     # Exactly SOURCE_MAX bytes were read; there may be more we couldn't fit
    call ErrHalt("plmmix-tangle: source too large\n");
  end
  return;
end
```

`InitSourceName` copies `argv[1]` (if present and non-null) into `SourceNameBuf` for use in `#line` directives; otherwise it uses the default name `stdin.plw`. The copy is bounded to `SOURCE_NAME_MAX - 1` bytes, leaving room for the null terminator. The final byte count is saved in `$src_name_len` so `EmitLineDirective` does not rescan the string on every chunk expansion.

⟨ ***InitSourceName procedure*** ⟩ ≡

```
def InitSourceName($argc, $argv):
  register $src, $dst, $limit, $ch, $tmp;
  $dst = $SourceNameBuf;
  $limit = SOURCE_NAME_MAX - 1;


  $src = &DefaultSourceName;
  goto ISUseDefault if $argc < 2;
   # argv is an octa array, so argv[1] lives 8 bytes past argv[0].
  $tmp = $argv + 8;
  $src = [$tmp].o;
  goto ISUseDefault if $src is zero;
  goto ISLoop;
  ISUseDefault:
  $src = &DefaultSourceName;
  ISLoop:
  goto ISDone if $limit is zero;
  $ch = [$src].bu;
  goto ISDone if $ch is zero;
  $tmp = $ch - '\\';
  goto ISStore if $tmp is nonzero;
  $ch = '/';
  ISStore:
  [$dst].bu = $ch;
  $dst = $dst + 1;
  $src = $src + 1;
  $limit = $limit - 1;
  goto ISLoop;
  ISDone:
  [$dst].bu = 0;
  $src_name_len = $dst - $SourceNameBuf;
  return;
```

```
end
```

8. String utilities.

Three small subroutines form the complete string library for this program. `StrLen` and `StrEq` serve `FindChunk`; `NormName` serves both the parser (when storing chunk-definition names) and the expander (when normalizing reference names for lookup).

The key invariant established by `NormName` is: two chunk names that differ only in the amount or kind of whitespace between tokens will normalize to the same byte string and therefore be treated as identical. For example, `@<Foo bar@>` and `@<Foo bar@>` refer to the same chunk. Whitespace here means any byte with value ≤ 32.

`StrLen` counts bytes up to (but not including) the first null byte.

⟨ ***String utilities*** ⟩ ≡

```
def StrLen($s) -> $n:
  register $c;
  $n = 0;
  SLLoop:
  $c = [$s].bu;
  if $c is zero:
    return $n;
  end
  $s = $s + 1;
  $n = $n + 1;
  goto SLLoop;
end
```

`StrEq` returns 1 if two byte strings of given lengths are equal, 0 otherwise. The fast path checks lengths first; only if they match are the bytes compared one by one.

⟨ ***String utilities*** ⟩ ≡

```
def StrEq($p1, $l1, $p2, $l2) -> $r:
  register $c1, $c2, $tmp;
  $tmp = $l1  cmpu $l2;
  if $tmp is nonzero:
    return 0;
  end
  SELoop:
  if $l1 is zero:
    return 1;
  end
  $c1 = [$p1].bu;
  $c2 = [$p2].bu;
  $tmp = $c1 - $c2;
  if $tmp is nonzero:
    return 0;
  end
  $p1 = $p1 + 1;
  $p2 = $p2 + 1;
  $l1 = $l1 - 1;
  goto SELoop;
end
```

`NormName` copies at most `$slen` bytes from `$src` into `$dst`, collapsing interior whitespace runs to a single space and stripping leading and trailing whitespace. Returns the normalized length.

The implementation uses a single flag `$prev_ws` that tracks whether the last emitted or skipped character was whitespace. It starts as 1 to suppress leading whitespace without a special case. The trailing-space trim at `NNDone` handles the one remaining case: a name that ends with whitespace, where `$prev_ws` would have already been set but one space byte was emitted.

⟨ ***String utilities*** ⟩ ≡

```
def NormName($src, $slen, $dst) -> $nlen:
  register $i, $ch, $prev_ws, $tmp;
  $i = 0;
  $nlen = 0;


  $prev_ws = 1;    # pretend leading whitespace already seen
  NNLoop:
  goto NNDone if $i >=u $slen;
  $ch = [$src].bu;
  $src = $src + 1;
  $i = $i + 1;
  $tmp = $ch  cmpu 32;
  goto NNWS if $tmp is nonpos;
   # Non-whitespace character
  $prev_ws = 0;
  [$dst].bu = $ch;
  $dst = $dst + 1;
  $nlen = $nlen + 1;
  goto NNLoop;
  NNWS:
  goto NNLoop if $prev_ws is nonzero;   # skip run
  $prev_ws = 1;
  [$dst].bu = 32;   # emit single space
  $dst = $dst + 1;
  $nlen = $nlen + 1;
  goto NNLoop;
  NNDone:
   # Trim trailing space, if any
  if $nlen is zero:
    return $nlen;
  end
  $tmp = $dst - 1;
  $ch = [$tmp].bu;
  $tmp = $ch  cmpu 32;
  if $tmp is nonzero:
    return $nlen;
  end
  return $nlen - 1;
end
```

9. File and include operations.

The include mechanism interns each source filename once, records that file id on every chunk entry, and loads included files into the tail of `SourceBuf` before parsing them recursively. A top-level include directive is recognized only when it occupies an entire code line, possibly with leading whitespace and an optional trailing comment. Each file is loaded at most once per build: re-including a file that has already been processed is silently ignored, so diamond-shaped include graphs are harmless. An include that points (directly or transitively) at a file currently on the active include stack is a hard error.

⟨ ***File and include operations*** ⟩ ≡

```
def InitFileTable():
  register $dst, $src, $left, $entry, $ch, $tmp;
  $tmp = $src_name_len + 1;
  $tmp = $tmp  cmpu FILE_NAME_BUF_SIZE;


  if $tmp is nonneg:
    call ErrHalt("plmmix-tangle: file name buffer overflow\n");
  end
  $dst = $FileNameBuf;
  $src = $SourceNameBuf;
  $left = $src_name_len;
  IFTLoop:
  goto IFTDone if $left is zero;
  $ch = [$src].bu;
  [$dst].bu = $ch;
  $dst = $dst + 1;
  $src = $src + 1;
  $left = $left - 1;
  goto IFTLoop;
  IFTDone:
  [$dst].bu = 0;
  $file_name_end = $src_name_len + 1;
  $entry = $FileTable;
  [$entry + FILE_NPTR].o = $FileNameBuf;
  [$entry + FILE_NLEN].o = $src_name_len;
  [$entry + FILE_STATE].o = FILE_STATE_LOADED;
  $file_cnt = 1;
  [$IncludeStack].o = 0;
  $include_depth = 1;
  return;
end
```

⟨ ***File and include operations*** ⟩ ≡

```
def FindFileName($nptr, $nlen) -> $idx:
  register $i, $entry, $cptr, $clen, $eq;
  $i = 0;
  $entry = $FileTable;
  FFLoop:
  if $i >=u $file_cnt:
    return -1;
  end
  $cptr = [$entry + FILE_NPTR].o;
  $clen = [$entry + FILE_NLEN].o;
  $eq = call StrEq($nptr, $nlen, $cptr, $clen);
  if $eq is nonzero:
```

```
      return $i;
    end
    $entry = $entry + FILE_ENTRY_SIZE;
    $i = $i + 1;
    goto FFLoop;
  end
```

⟨ ***File and include operations*** ⟩ ≡

```
  def InternFileName($nptr, $nlen) -> $idx:
    register $entry, $dst, $src, $left, $tmp, $ch;
    $idx = call FindFileName($nptr, $nlen);
    goto IFNAdd if $idx is neg;


    return $idx;
    IFNAdd:
    $tmp = $file_cnt  cmpu FILE_MAX;
    if $tmp is nonneg:
      call ErrHalt("plmmix-tangle: too many included files\n");
    end
    $tmp = $file_name_end + $nlen;
    $tmp = $tmp + 1;
    $tmp = $tmp  cmpu FILE_NAME_BUF_SIZE;
    if $tmp is nonneg:
      call ErrHalt("plmmix-tangle: file name buffer overflow\n");
    end
    $idx = $file_cnt;
    $entry = $idx * FILE_ENTRY_SIZE;
    $entry = $FileTable + $entry;
    $dst = $FileNameBuf + $file_name_end;
    [$entry + FILE_NPTR].o = $dst;
    [$entry + FILE_NLEN].o = $nlen;
    [$entry + FILE_STATE].o = 0;
    $src = $nptr;
    $left = $nlen;
    IFNCopy:
    goto IFNDone if $left is zero;
    $ch = [$src].bu;
    [$dst].bu = $ch;
    $src = $src + 1;
    $dst = $dst + 1;
    $left = $left - 1;
    goto IFNCopy;
    IFNDone:
    [$dst].bu = 0;
    $file_name_end = $file_name_end + $nlen;
    $file_name_end = $file_name_end + 1;
    $file_cnt = $file_cnt + 1;
    return $idx;
  end
```

⟨ ***File and include operations*** ⟩ ≡

```
def FileIsLoaded($fid) -> $loaded:
  register $entry;
  $entry = $fid * FILE_ENTRY_SIZE;
  $entry = $FileTable + $entry;
  $loaded = [$entry + FILE_STATE].o;
  return $loaded;
end
```

⟨ ***File and include operations*** ⟩ ≡

```
def PushIncludeFile($fid):
  register $tmp, $slot;
  $tmp = $include_depth  cmpu MAX_DEPTH;
  if $tmp is nonneg:
    call ErrHalt("plmmix-tangle: include cycle detected\n");
  end
  $slot = $include_depth * 8;
  $slot = $IncludeStack + $slot;
  [$slot].o = $fid;
  $include_depth = $include_depth + 1;
  return;
end
```

⟨ ***File and include operations*** ⟩ ≡

```
def PopIncludeFile():
  if $include_depth is zero:
    return;
  end
  $include_depth = $include_depth - 1;
  return;
end
```

⟨ ***File and include operations*** ⟩ ≡

```
def FileOnIncludeStack($fid) -> $found:
  register $i, $slot, $cur, $tmp;
  $found = 0;
  $i = 0;
  FOISLoop:
  if $i >=u $include_depth:
    return $found;
  end
  $slot = $i * 8;
  $slot = $IncludeStack + $slot;
  $cur = [$slot].o;
  $tmp = $cur - $fid;
  if $tmp is zero:
    return 1;
  end
  $i = $i + 1;
  goto FOISLoop;
end
```

⟨ ***File and include operations*** ⟩ ≡

```
def ResolveIncludePath($owner_fid, $rel_ptr, $rel_len, $dst) -> $out_len:
  register $entry, $src, $src_len, $i, $copy_len, $ch, $tmp;
  $entry = $owner_fid * FILE_ENTRY_SIZE;
  $entry = $FileTable + $entry;


  $src = [$entry + FILE_NPTR].o;
  $src_len = [$entry + FILE_NLEN].o;
  $out_len = 0;
  goto RIPAbsolute if $rel_len is zero;
  $ch = [$rel_ptr].bu;
  $tmp = $ch - '\\';
  goto RIPSlash if $tmp is zero;
  $tmp = $ch - '/';
  goto RIPSlash if $tmp is zero;
  goto RIPDrive;
  RIPAbsolute:
  RIPSlash:
  $src = $rel_ptr;
  $src_len = $rel_len;
  goto RIPCopyTail;
  RIPDrive:
  goto RIPRelative if $rel_len < 2;
  $ch = [$rel_ptr + 1].bu;
  $tmp = $ch - ':';
  goto RIPRelative if $tmp is nonzero;
  $src = $rel_ptr;
  $src_len = $rel_len;
  goto RIPCopyTail;
  RIPRelative:
  $i = $src_len;
  RIPFindBase:
  goto RIPBaseDone if $i is zero;
  $i = $i - 1;
  $ch = [$src + $i].bu;
  $tmp = $ch - '\\';
  goto RIPBaseSlash if $tmp is zero;
  $tmp = $ch - '/';
  goto RIPFindBase if $tmp is nonzero;
  RIPBaseSlash:
  $i = $i + 1;
  RIPBaseDone:
  $copy_len = $i + $rel_len;
  $copy_len = $copy_len + 1;
  $tmp = $copy_len  cmpu PATH_BUF_SIZE;
  if $tmp is nonneg:
    call ErrHalt("plmmix-tangle: include path too long\n");
  end
  RIPBaseCopy:
  goto RIPBaseCopied if $i is zero;
  $ch = [$src].bu;
  $tmp = $ch - '\\';
  goto RIPStoreBase if $tmp is nonzero;
  $ch = '/';
  RIPStoreBase:
  [$dst].bu = $ch;
  $dst = $dst + 1;
  $src = $src + 1;
  $i = $i - 1;
  $out_len = $out_len + 1;
  goto RIPBaseCopy;
```

```
  RIPBaseCopied:
  $src = $rel_ptr;
  $src_len = $rel_len;
  RIPCopyTail:
  $copy_len = $out_len + $src_len;
  $copy_len = $copy_len + 1;
  $tmp = $copy_len  cmpu PATH_BUF_SIZE;
  if $tmp is nonneg:
    call ErrHalt("plmmix-tangle: include path too long\n");
  end
  RIPTailLoop:
  goto RIPDone if $src_len is zero;
  $ch = [$src].bu;
  $tmp = $ch - '\\';
  goto RIPStoreTail if $tmp is nonzero;
  $ch = '/';
  RIPStoreTail:
  [$dst].bu = $ch;
  $dst = $dst + 1;
  $src = $src + 1;
  $src_len = $src_len - 1;
  $out_len = $out_len + 1;
  goto RIPTailLoop;
  RIPDone:
  [$dst].bu = 0;
  return $out_len;
end
```

⟨ ***File and include operations*** ⟩ ≡

```
def ReadFileIntoSource($path_ptr):
  register $tmp, $avail;
  $tmp = $FOpenArgs;
  [$tmp].o = $path_ptr;


  [$tmp + 8].o = TEXT_READ_MODE;
  $out = $tmp;
  trap 0, FOPEN, 3;
  if $out is neg:
    call ErrHalt("plmmix-tangle: unable to open include file\n");
  end
  $read_bptr = $SourceBuf + $src_len;
  $avail = SOURCE_MAX - $src_len;
  if $avail is zero:
    call ErrHalt("plmmix-tangle: source too large\n");
  end
  $tmp = $FReadArgs;
  [$tmp].o = $read_bptr;
  [$tmp + 8].o = $avail;
  $out = $tmp;
  trap 0, FREAD, 3;
  $read_blen = $out + $avail;
  goto RFISizeOk if $read_blen is nonneg;
  call ErrHalt("plmmix-tangle: source too large\n");
  RFISizeOk:
  $tmp = $avail;
  if $read_blen >=u $tmp:
    call ErrHalt("plmmix-tangle: source too large\n");
  end
```

```
  $src_len = $src_len + $read_blen;
  trap 0, FCLOSE, 3;
  return;
end
```

⟨ ***File and include operations*** ⟩ ≡

```
def LoadIncludedFile($owner_fid, $path_ptr, $path_len):
  register $fid, $tmp, $entry;
  $path_len = call ResolveIncludePath($owner_fid, $path_ptr, $path_len, $ResolvedPathBuf);
  $fid = call InternFileName($ResolvedPathBuf, $path_len);
  $tmp = call FileOnIncludeStack($fid);
  if $tmp is nonzero:
    call ErrHalt("plmmix-tangle: include cycle detected\n");
  end
  $tmp = call FileIsLoaded($fid);
  if $tmp is nonzero:
    return;             # already included earlier in the build; silently skip
  end
  $entry = $fid * FILE_ENTRY_SIZE;
  $entry = $FileTable + $entry;
  [$entry + FILE_STATE].o = FILE_STATE_LOADED;
  call PushIncludeFile($fid);
  call ReadFileIntoSource($ResolvedPathBuf);
  call ParsePLW($read_bptr, $read_blen, $fid, $ResolvedPathBuf, $path_len);
  call PopIncludeFile();
  return;
end
```

⟨ ***File and include operations*** ⟩ ≡

```
def MatchIncludeLine($scan, $send, $path_dst) -> $matched:
  register $cur, $ch, $tmp;
  $matched = 0;
  $match_next = $scan;


  $match_path_len = 0;
  $match_line_inc = 0;
  $cur = $scan;
  MILSkipWS:
  goto MILNo if $cur >=u $send;
  $ch = [$cur].bu;
  $tmp = $ch - ' ';
  goto MILStepWS if $tmp is zero;
  $tmp = $ch - '\t';
  goto MILStepWS if $tmp is zero;
  goto MILKeyword;
  MILStepWS:
  $cur = $cur + 1;
  goto MILSkipWS;
  MILKeyword:
  $tmp = $send - $cur;
  goto MILNo if $tmp < 7;
  $ch = [$cur].bu;       $tmp = $ch - 'i'; goto MILNo if $tmp is nonzero;
  $ch = [$cur + 1].bu;   $tmp = $ch - 'n'; goto MILNo if $tmp is nonzero;
  $ch = [$cur + 2].bu;   $tmp = $ch - 'c'; goto MILNo if $tmp is nonzero;
  $ch = [$cur + 3].bu;   $tmp = $ch - 'l'; goto MILNo if $tmp is nonzero;
```

```
$ch = [$cur + 4].bu;  $tmp = $ch - 'u'; goto MILNo if $tmp is nonzero;
$ch = [$cur + 5].bu;  $tmp = $ch - 'd'; goto MILNo if $tmp is nonzero;
$ch = [$cur + 6].bu;  $tmp = $ch - 'e'; goto MILNo if $tmp is nonzero;
$cur = $cur + 7;
goto MILMalformed if $cur >=u $send;
$ch = [$cur].bu;
$tmp = $ch  cmpu 32;
goto MILGap if $tmp is nonpos;
goto MILMalformed;
MILGap:
goto MILMalformed if $cur >=u $send;
$ch = [$cur].bu;
$tmp = $ch - ' ';
goto MILGapStep if $tmp is zero;
$tmp = $ch - '\t';
goto MILGapStep if $tmp is zero;
goto MILQuote;
MILGapStep:
$cur = $cur + 1;
goto MILGap;
MILQuote:
$tmp = $ch - 34;
goto MILMalformed if $tmp is nonzero;
$cur = $cur + 1;
MILPath:
goto MILMalformed if $cur >=u $send;
$ch = [$cur].bu;
$tmp = $ch - 34;
goto MILAfterQuote if $tmp is zero;
$tmp = $ch - '\n';
goto MILMalformed if $tmp is zero;
$tmp = $match_path_len + 1;
$tmp = $tmp  cmpu PATH_BUF_SIZE;
if $tmp is nonneg:
  call ErrHalt("plmmix-tangle: include path too long\n");
end
$path_dst[$match_path_len].bu = $ch;
$match_path_len = $match_path_len + 1;
$cur = $cur + 1;
goto MILPath;
MILAfterQuote:
$cur = $cur + 1;
MILAfterWS:
goto MILMalformed if $cur >=u $send;
$ch = [$cur].bu;
$tmp = $ch - ' ';
goto MILAfterStep if $tmp is zero;
$tmp = $ch - '\t';
goto MILAfterStep if $tmp is zero;
goto MILSemi;
MILAfterStep:
$cur = $cur + 1;
goto MILAfterWS;
MILSemi:
$tmp = $ch - ';';
goto MILMalformed if $tmp is nonzero;
$cur = $cur + 1;
MILTail:
goto MILDone if $cur >=u $send;
$ch = [$cur].bu;
$tmp = $ch - ' ';
goto MILTailStep if $tmp is zero;
```

```
  $tmp = $ch - '\t';
  goto MILTailStep if $tmp is zero;
  $tmp = $ch - ' #';
  goto MILComment if $tmp is zero;
  $tmp = $ch - '\n';
  goto MILDoneNL if $tmp is zero;
  goto MILMalformed;
  MILTailStep:
  $cur = $cur + 1;
  goto MILTail;
  MILComment:
  $cur = $cur + 1;
  MILCommentLoop:
  goto MILDone if $cur >=u $send;
  $ch = [$cur].bu;
  $tmp = $ch - '\n';
  goto MILDoneNL if $tmp is zero;
  $cur = $cur + 1;
  goto MILCommentLoop;
  MILDoneNL:
  $cur = $cur + 1;
  $match_line_inc = 1;
  MILDone:
  $path_dst[$match_path_len].bu = 0;
  $matched = 1;
  $match_next = $cur;
  return $matched;
  MILMalformed:
  call ErrHalt("plmmix-tangle: malformed include directive\n");
  MILNo:
  return $matched;
end
```

10. Chunk table operations.

The chunk table is a flat array of `CHUNK_MAX` fixed-size 40-byte entries (as declared in the constants section). Multiple entries may share the same normalized name, arising from repeated append definitions (same name, multiple bodies); `ExpandChunk` iterates over *all* matching entries in definition order, concatenating their bodies.

Because chunk bodies are slices of the immutable `SourceBuf`, neither `FindChunk` nor `AddChunk` copies any body text. The only memory consumed per definition beyond the entry itself is the normalized name appended to `NameBuf`.

`FindChunk` returns the 0-based index of the *first* entry whose normalized name matches the argument, or -1 if no match is found. Callers that want all matching entries (i.e., `ExpandChunk`) scan the table themselves using this index as a starting point.

⟨ ***Chunk table operations*** ⟩ ≡

```
def FindChunk($nptr, $nlen) -> $idx:
  register $i, $entry, $cptr, $clen, $eq;
  $i = 0;
  $entry = $ChunkTable;
  FCLoop:
  if $i >=u $chunk_cnt:
    return -1;
  end
  $cptr = [$entry].o;                # CH_NPTR = 0
  $clen = [$entry + CH_NLEN].o;
  $eq = call StrEq($nptr, $nlen, $cptr, $clen);
  if $eq is nonzero:
    return $i;
  end
  $entry = $entry + CHUNK_ENTRY_SIZE;
  $i = $i + 1;
  goto FCLoop;
end
```

`AddChunk` appends one entry to the chunk table. The caller is responsible for having already normalized the name and appended it to `NameBuf`; the `$nptr` and `$nlen` arguments point into `NameBuf`. The body pointer `$bptr` points into `SourceBuf`.

⟨ ***Chunk table operations*** ⟩ ≡

```
def AddChunk($nptr, $nlen, $bptr, $blen, $line, $fptr, $flen):
  register $entry, $tmp;
  $tmp = $chunk_cnt  cmpu CHUNK_MAX;
  if $tmp is nonneg:
    call ErrHalt("plmmix-tangle: too many chunk definitions\n");
  end
  $entry = $chunk_cnt * CHUNK_ENTRY_SIZE;
  $entry = $entry + $ChunkTable;
  [$entry].o             = $nptr;      # CH_NPTR
  [$entry + CH_NLEN].o   = $nlen;
  [$entry + CH_BPTR].o   = $bptr;
  [$entry + CH_BLEN].o   = $blen;
  [$entry + CH_LINE].o   = $line;
  [$entry + CH_FPTR].o   = $fptr;
  [$entry + CH_FLEN].o   = $flen;
  $chunk_cnt = $chunk_cnt + 1;
  return;
end
```

11. Parsing the literate web source.

`ParsePLW` makes one forward pass over `SourceBuf`, recognizing the four active sigils and building the chunk table. It is a two-state machine:

State 0 — prose. Bytes are silently ignored. The parser watches for an `@` sigil that begins a code chunk definition.

State 1 — code. Bytes advance the body-end marker `$bend`. A new chunk-definition sigil or a prose-section sigil flushes the current chunk to the table and either starts a new chunk or returns to prose state.

The body of each chunk is represented as a *pointer range* [*bstart*, *bend*) into `SourceBuf`. No body text is copied; `AddChunk` records only the two pointers.

Tangling note. The eight subchunks that follow share a single flat label namespace (`PMain`, `PAt`, `PDone`, `PChunkBracket`, ...): jumps cross subchunk boundaries freely. They must be tangled in the order listed in `@<ParsePLW procedure@>=`, which is the order shown below. An LLM or human reading one subchunk in isolation will see labels resolved by sibling subchunks; treat the eight subchunks as one procedure split for narrative purposes only. The subchunk names use "at-sign" rather than a literal at-sign where practical, so the headings remain safe to edit without accidentally introducing an `@>` terminator.

The key state variables are:

`$scan`, `$send` Read cursor and one-past-end of `SourceBuf`.

`$state` 0 for prose, 1 for code.

`$bstart`, `$bend` Start and current end of the open chunk body, both pointing into `SourceBuf`. In code state, every non-`@` byte advances `$bend` to `$scan`.

`$cur_nptr`, `$cur_nlen`, `$cur_line` Name pointer, name length, and definition-line of the chunk currently being accumulated.

`$line_no` Current source line (1-based), incremented on every newline byte. Stored in `CH_LINE` and used for `#line` directives.

The variable group `$ns`, `$nr`, `$nlen` is scratch used only while scanning a chunk name between `@<` and `@>`.

⟨ ***ParsePLW procedure*** ⟩ ≡

```
def ParsePLW($src_ptr, $src_len, $file_id, $file_ptr, $file_len):
  register $scan, $send, $state, $bstart, $bend;
  register $cur_nptr, $cur_nlen, $cur_line;
  register $ch, $next, $tmp, $line_no, $at_line;


  register $ns, $nr, $nlen;    # name scratch: start, raw len, normalized len
  register $line_start, $inc_match, $inc_next, $inc_len, $inc_lines;
  ⟨ParsePLW: initialize scan state⟩
  ⟨ParsePLW: main scan loop⟩
  PAt:
   # $scan now points just past the '@'
  $at_line = $line_no;
  if $scan >=u $send:
     # A bare '@' at end of file is always a truncated sigil.
    call ErrHalt("plmmix-tangle: source ends with bare @@\n");
  end
  $next = [$scan].bu;

   # @@@@ --- literal @@
  $tmp = $next - '@';
  goto PAtAt if $tmp is zero;

   # @@* or @@<whitespace> --- prose section
  $tmp = $next - '*';
  goto PStartProse if $tmp is zero;
  $tmp = $next  cmpu 32;
  goto PStartProse if $tmp is nonpos;

   # @@< --- chunk reference or definition
  $tmp = $next - '<';
```

```
    goto PChunkBracket if $tmp is zero;

     # Other @@x --- treat @@ as literal in code, skip in prose
    goto PMain if $state is zero;
    $bend = $scan;     # '@' was at $scan-1; body extends to include it
    $line_start = 0;
    goto PMain;

    PAtAt:
     # @@ in code is part of the body (ExpandBody handles the escaping)
    $scan = $scan + 1;     # skip second @@
    goto PMain if $state is zero;
    $bend = $scan;
    $line_start = 0;
    goto PMain;
    ⟨ParsePLW: transition to prose state⟩
    ⟨ParsePLW: process chunk bracket⟩
    PDone:
    ⟨ParsePLW: flush final chunk and return⟩
end
```

Initialization sets `$scan` to the start of `SourceBuf` and `$send` to one past the last valid byte. Both `$bstart` and `$bend` begin at the same position; they diverge when the first code chunk opens.

⟨***ParsePLW: initialize scan state***⟩ ≡

```
$scan    = $src_ptr;
$send    = $src_ptr + $src_len;
$state   = 0;
$bstart = $src_ptr;
$bend    = $src_ptr;
$cur_nptr = 0;
$cur_nlen = 0;
$cur_line = 0;
$line_no = 1;
$line_start = 1;
```

In the main loop each byte falls into one of three cases: the `@` sigil (jump to `PAt`), a newline (increment `$line_no`), or any other byte. In prose state other bytes are discarded. In code state they advance `$bend` to include the current byte in the open body.

⟨***ParsePLW: main scan loop***⟩ ≡

```
PMain:
goto PDone if $scan >=u $send;
goto PReadByte if $state is zero;
goto PReadByte if $line_start is zero;


$inc_match = call MatchIncludeLine($scan, $send, $IncludePathBuf);
goto PReadByte if $inc_match is zero;
$inc_len = $match_path_len;
$inc_lines = $match_line_inc;
$tmp = $scan - $bstart;
goto PIncNoFlush if $tmp is zero;
call AddChunk($cur_nptr, $cur_nlen, $bstart, $tmp, $cur_line, $file_ptr, $file_len);
PIncNoFlush:
```

```
$scan = $match_next;
call LoadIncludedFile($file_id, $IncludePathBuf, $inc_len);
$bstart = $scan;
$bend = $scan;
$line_no = $line_no + $inc_lines;
$line_start = 1;
goto PMain;
PReadByte:
$ch = [$scan].bu;
$scan = $scan + 1;
$tmp = $ch - '@';
goto PAt if $tmp is zero;
 # Regular character
goto PRegular if $state is zero;
$bend = $scan;
PRegular:
$tmp = $ch - '\n';
goto PRegularNoNL if $tmp is nonzero;
$line_no = $line_no + 1;
$line_start = 1;
goto PMain;
PRegularNoNL:
$line_start = 0;
goto PMain;
```

After seeing `@`, the parser reads one more byte to classify the sigil. There are four cases: the at-at escape (emits a literal `@`), at-star or whitespace (starts a prose section), at-angle (opens a chunk name bracket), and anything else (treat `@` as literal in code, ignore in prose).

⟨ ***ParsePLW: handle sigil dispatch*** ⟩ ≡

```
PAt:
 # $scan now points just past the '@'
$at_line = $line_no;
if $scan >=u $send:


   # A bare '@' at end of file is always a truncated sigil.
  call ErrHalt("plmmix-tangle: source ends with bare @@\n");
end
$next = [$scan].bu;

 # @@@@ - literal @@
$tmp = $next - '@';
goto PAtAt if $tmp is zero;

 # @@* or @@<whitespace> - prose section
$tmp = $next - '*';
goto PStartProse if $tmp is zero;
$tmp = $next  cmpu 32;
goto PStartProse if $tmp is nonpos;

 # @@< - chunk reference or definition
$tmp = $next - '<';
goto PChunkBracket if $tmp is zero;

 # Other @@x - treat @@ as literal in code, skip in prose
goto PMain if $state is zero;
$bend = $scan;     # '@' was at $scan-1; body extends to include it
$line_start = 0;
```

```
goto PMain;

PAtAt:
 # @@ in code is part of the body (ExpandBody handles the escaping)
$scan = $scan + 1;     # skip second @@
goto PMain if $state is zero;
$bend = $scan;
$line_start = 0;
goto PMain;
```

When a prose-section sigil (at-star or at-whitespace) is seen, the parser flushes the currently open chunk (if any) to the chunk table before entering prose state. The body length is `$bend - $bstart`; a zero-length body is valid. After flushing, at-star skips the rest of the heading line so that the section title does not accidentally resemble a sigil.

⟨ ***ParsePLW: transition to prose state*** ⟩ ≡

```
PStartProse:
 # Transition to prose state; flush any open chunk
goto PFlushDone if $state is zero;
goto PFlushDone if $cur_nlen is zero;
$tmp = $bend - $bstart;     # body length
call AddChunk($cur_nptr, $cur_nlen, $bstart, $tmp, $cur_line, $file_ptr, $file_len);
PFlushDone:
$state = 0;
$line_start = 0;
 # Skip the rest of the heading line (for @@*)
$tmp = $next - '*';
goto PMain if $tmp is nonzero;
PSkipHeading:
goto PMain if $scan >=u $send;
$ch = [$scan].bu;
$scan = $scan + 1;
$tmp = $ch - '\n';
goto PHeadingNL if $tmp is zero;
goto PSkipHeading;
PHeadingNL:
$line_no = $line_no + 1;
goto PMain;
```

When `@<` is seen, the parser scans forward to find the closing `@>`. The raw name occupies bytes from `$ns` up to but not including the two closing sigil bytes. The byte immediately after `@>` determines whether this is a definition (the closing sigil followed by `=`) or a reference (no `=`).

For a *definition*: flush any open chunk, normalize the name into `NameBuf`, set `$bstart` just after the `=` (skipping an optional newline), and enter code state.

For a *reference* inside code: include the full sigil text in the current body by advancing `$bend`; `ExpandBody` will recurse into it later. References in prose state are silently ignored.

⟨ ***ParsePLW: process chunk bracket*** ⟩ ≡

```
PChunkBracket:
$scan = $scan + 1;     # skip '<'
$ns = $scan;           # name starts here
PFindClose:


goto PUnclosed if $scan >=u $send;
$ch = [$scan].bu;
$scan = $scan + 1;
$tmp = $ch - '\n';
goto PNameNoNL if $tmp is nonzero;
$line_no = $line_no + 1;
PNameNoNL:
$tmp = $ch - '@';
goto PFindClose if $tmp is nonzero;
 # Saw '@'; peek at next byte WITHOUT advancing $scan
goto PUnclosed if $scan >=u $send;
$ch = [$scan].bu;
 # Two consecutive at-signs inside a name: consume the second one and continue.
 # Without this, @@> is misread as the closing sigil.
$tmp = $ch - '@';
goto PNameAtAt if $tmp is zero;
$tmp = $ch - '>';
goto PFindClose if $tmp is nonzero;
 # Found @>; name is [$ns .. $scan-2)
$scan = $scan + 1;     # skip '>'
$nr = $scan - $ns;     # distance from name start to scan
$nr = $nr - 2;         # subtract 2 for the @> sigil bytes

 # Is the next character '='?
goto PChunkRef if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - '=';
goto PChunkRef if $tmp is nonzero;

 # --- Chunk definition (at-angle name at-angle-equals) ---
$scan = $scan + 1;     # skip '='
 # Skip optional newline immediately after '='
goto PNoNL if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - '\n';
goto PNoNL if $tmp is nonzero;
$scan = $scan + 1;
$line_no = $line_no + 1;
PNoNL:
 # Flush previous chunk if open
goto PNoFlush if $state is zero;
goto PNoFlush if $cur_nlen is zero;
$tmp = $bend - $bstart;
call AddChunk($cur_nptr, $cur_nlen, $bstart, $tmp, $cur_line, $file_ptr, $file_len);
PNoFlush:
 # Normalize and store name in NameBuf.
 # NormName can write at most $nr bytes (normalization only shrinks), so check
 # $name_end + $nr against the buffer limit rather than $name_end alone.
$tmp = $name_end + $nr;
$tmp = $tmp  cmpu NAME_BUF_SIZE;
if $tmp is nonneg:
  call ErrHalt("plmmix-tangle: name buffer overflow\n");
end
$tmp = $NameBuf + $name_end;
$nlen = call NormName($ns, $nr, $tmp);
```

```
$cur_nptr = $tmp;
$cur_nlen = $nlen;
$cur_line = $line_no;    # $line_no already accounts for the optional newline after '='
$name_end = $name_end + $nlen;
$bstart = $scan;
$bend   = $scan;
$state  = 1;
$line_start = 1;
goto PMain;

PNameAtAt:
$scan = $scan + 1;     # consume second at-sign, then resume name scan
goto PFindClose;

PChunkRef:
 # chunk-ref inside code: body includes the whole sigil text
goto PMain if $state is zero;
$bend = $scan;
$line_start = 0;
goto PMain;

PUnclosed:
call ErrHalt("plmmix-tangle: unclosed @@< in source\n");
```

At end of input, any chunk still open is flushed. A zero-length final chunk (a definition marker immediately followed by end-of-file) is still recorded, since an empty body is a valid definition.

⟨ ***ParsePLW: flush final chunk and return*** ⟩ ≡

```
if $state is nonzero:
  if $cur_nlen is nonzero:
    $tmp = $bend - $bstart;
    call AddChunk($cur_nptr, $cur_nlen, $bstart, $tmp, $cur_line, $file_ptr, $file_len);
  end
end
return;
```

12. Expanding chunks.

Expansion is driven by three mutually recursive procedures.

`ExpandRoot` chooses the starting point: if a chunk named `Program` exists it expands that alone; otherwise it iterates over the chunk table and expands each unique name exactly once in definition order. This fallback mode lets small `.plw` files omit the `@<Program@>` root.

`ExpandChunk` is the indirection layer: given a name, it finds *all* table entries with that name and calls `ExpandBody` on each in sequence, implementing the append semantics of multiple definitions sharing the same name.

`ExpandBody` does the byte-by-byte scan of a raw body slice. Regular bytes and `@` escapes are emitted directly. Every chunk-reference sigil triggers a recursive call to `ExpandChunk`, and a `#line` directive is emitted after each such call to reset the compiler's notion of the current source position.

A depth counter passed through the call chain limits recursion to `MAX_DEPTH` levels, catching cycles before the stack overflows. For example, if chunk A references chunk B and B references A,

the counter reaches `MAX_DEPTH` and `ErrHalt` terminates the program with the cycle error message.

⟨ ***Expansion procedures*** ⟩ ≡

```
⟨ ExpandRoot procedure ⟩
⟨ ExpandChunk procedure ⟩
⟨ ExpandBody procedure ⟩
```

⟨ ***ExpandRoot procedure*** ⟩ ≡

```
def ExpandRoot():
  register $rptr, $rlen, $idx, $i, $entry, $np, $nl, $found, $tmp;
   # Look for a chunk named "Program"
  $rptr = &RootChunk;


  $rlen = ROOT_LEN;
  $idx = call FindChunk($rptr, $rlen);
  goto ERConcat if $idx is neg;
   # Found: expand it
  call ExpandChunk($rptr, $rlen, 0);
  return;
  ERConcat:
   # Concatenation mode: expand each unique name once in definition order
  $i = 0;
  $entry = $ChunkTable;
  ERCLoop:
  if $i >=u $chunk_cnt:
    return;
  end
  $np = [$entry].o;
  $nl = [$entry + CH_NLEN].o;
   # FindChunk must return some index here because this entry's own name matches.
   # If an earlier entry has the same name, skip it as already expanded.
  $found = call FindChunk($np, $nl);
  goto ERCNext if $found <u $i;
  call ExpandChunk($np, $nl, 0);
  ERCNext:
  $entry = $entry + CHUNK_ENTRY_SIZE;
  $i = $i + 1;
  goto ERCLoop;
end
```

`ExpandChunk` finds all entries for the given name and expands their bodies in order. A chunk that is referenced but never defined is a fatal error: there is no way to produce valid output in that case.

⟨ ***ExpandChunk procedure*** ⟩ ≡

```
def ExpandChunk($nptr, $nlen, $depth):
  register $i, $entry, $cptr, $clen, $bptr, $blen, $eq, $found, $tmp, $line, $file_ptr, $file_len;
  $tmp = $depth  cmpu MAX_DEPTH;
  if $tmp is nonneg:


    call ErrHalt("plmmix-tangle: chunk cycle detected\n");
  end
  $found = 0;
  $i = 0;
  $entry = $ChunkTable;
  ECLoop:
  goto ECDone if $i >=u $chunk_cnt;
  $cptr = [$entry].o;
  $clen = [$entry + CH_NLEN].o;
  $eq = call StrEq($nptr, $nlen, $cptr, $clen);
  goto ECNext if $eq is zero;
  $found = 1;
  $bptr = [$entry + CH_BPTR].o;
  $blen = [$entry + CH_BLEN].o;
  $line = [$entry + CH_LINE].o;
  $file_ptr = [$entry + CH_FPTR].o;
  $file_len = [$entry + CH_FLEN].o;
  call EmitLineDirective($cptr, $clen, $file_ptr, $file_len, $line);
  $tmp = $depth + 1;
  call ExpandBody($cptr, $clen, $file_ptr, $file_len, $bptr, $blen, $tmp, $line);
  ECNext:
  $entry = $entry + CHUNK_ENTRY_SIZE;
  $i = $i + 1;
  goto ECLoop;
  ECDone:
  if $found is nonzero:
    return;
  end
  call ErrHalt("plmmix-tangle: referenced chunk never defined\n");
  return;
end
```

ExpandBody scans a raw body slice byte by byte. The invariant is that `$scan` is always a valid address in the range `[$bptr .. $end)` throughout the procedure.

The body text was recorded verbatim from `SourceBuf`, including any chunk-reference sigils. The expander re-implements the same sigil recognizer as the parser, but only for two sigils: `@@` (emit one literal `@`) and the chunk-reference sigil (recurse into ExpandChunk). All other `@x` sequences are emitted literally, since the parser already ensured that any such bytes in a code body are intentional.

After each such inline expansion, a fresh `#line` directive names the current chunk at the line *following* the reference sigil. If the reference name itself spans lines, the resumed line number must account for those embedded newlines as well. This keeps the downstream compiler's error messages anchored to the correct source location.

⟨ ***ExpandBody procedure*** ⟩ ≡

```
def ExpandBody($owner_nptr, $owner_nlen, $owner_file_ptr, $owner_file_len, $bptr, $blen, $depth,
  $base_line):
  register $scan, $end, $ch, $next, $tmp, $line;
  register $ns, $nr, $nbuf, $nlen;     # name parsing temporaries
  $scan = $bptr;


  $end  = $bptr + $blen;
  $line = $base_line;
  EBMain:
  if $scan >=u $end:
    return;
  end
  $ch = [$scan].bu;
  $scan = $scan + 1;
  $tmp = $ch - '@';
  goto EBAt if $tmp is zero;
  call AppendByte($ch);
  $tmp = $ch - '\n';
  goto EBMain if $tmp is nonzero;
  $line = $line + 1;
  goto EBMain;
  EBAt:
   # '@' seen; check what follows
  if $scan >=u $end:
    return;
  end
  $next = [$scan].bu;
   # @@@@ -> emit single @@
  $tmp = $next - '@';
  goto EBAtAt if $tmp is zero;
   # @@< -> chunk reference
  $tmp = $next - '<';
  goto EBRef if $tmp is zero;
   # Other @x -> emit the '@' literally and re-process x
  call AppendByte('@');
  goto EBMain;
  EBAtAt:
  $scan = $scan + 1;
  call AppendByte('@');
  goto EBMain;
  EBRef:
  $scan = $scan + 1;     # skip '<'
  $ns = $scan;           # name starts here
  EBFindClose:
   # Each iteration reads one byte ($scan++) then peeks at the next WITHOUT
   # advancing, so a mismatch on the second byte re-reads it cleanly next time.
  goto EBUnclosed if $scan >=u $end;
  $ch = [$scan].bu;
  $scan = $scan + 1;
  $tmp = $ch - '\n';
  goto EBRefNameNoNL if $tmp is nonzero;
  $line = $line + 1;
  EBRefNameNoNL:
  $tmp = $ch - '@';
  goto EBFindClose if $tmp is nonzero;
   # Saw '@'; peek at next byte WITHOUT advancing $scan
  goto EBUnclosed if $scan >=u $end;
  $ch = [$scan].bu;
   # Two consecutive at-signs inside a name: consume the second one and continue.
```

```
    # This prevents @@> from being misread as the closing sigil (mirrors PNameAtAt).
  $tmp = $ch - '@';
  goto EBRefNameAtAt if $tmp is zero;
  $tmp = $ch - '>';
  goto EBFindClose if $tmp is nonzero;
  $scan = $scan + 1;     # skip '>'
  $nr = $scan - $ns;     # distance from name start to scan
  $nr = $nr - 2;         # subtract 2 for the @> sigil bytes
    # Normalize name into NameBuf for lookup.
    # NormName writes at most $nr bytes, so bound against $name_end + $nr.
  $tmp = $name_end + $nr;
  $tmp = $tmp  cmpu NAME_BUF_SIZE;
  if $tmp is nonneg:
    call ErrHalt("plmmix-tangle: name buffer overflow\n");
  end
  $nbuf = $NameBuf + $name_end;
  $nlen = call NormName($ns, $nr, $nbuf);
  $name_end = $name_end + $nlen;
  call ExpandChunk($nbuf, $nlen, $depth);
    # Resume at the current line in the owner body ($line, not $line+1, because
    # #line N means "the following byte is on line N" and we haven't advanced past
    # the line containing @>).
  call EmitLineDirective($owner_nptr, $owner_nlen, $owner_file_ptr, $owner_file_len, $line);
  goto EBMain;
  EBRefNameAtAt:
  $scan = $scan + 1;     # consume second at-sign, then resume name scan
  goto EBFindClose;
  EBUnclosed:
  call ErrHalt("plmmix-tangle: unclosed @@< in source\n");
end
```

13. Output buffer and \#line directives.

All output accumulates in `OutputBuf` via `AppendByte`, the hot path called for every non-sigil byte in every chunk body. `AppendBytes` and `AppendCStr` are convenience wrappers used by `EmitLineDirective`. `AppendInt` encodes a non-negative integer as decimal ASCII, also for `EmitLineDirective`.

`EmitLineDirective` emits a directive in the format:

```
#line "chunk-name" "source-file" line-number
```

This PL/MMIX extension lets the compiler map object-code addresses back to literate-source locations in error messages and debugger output. It is emitted once before each chunk body (by `ExpandChunk`) and once after each inline chunk expansion (by `ExpandBody`).

⟨ ***Output procedures*** ⟩ ≡

```
def AppendByte($ch):
  register $tmp;
  $tmp = $out_end  cmpu OUT_MAX;
  if $tmp is nonneg:
    call ErrHalt("plmmix-tangle: output buffer full\n");
  end
  $OutputBuf[$out_end].bu = $ch;
  $out_end = $out_end + 1;
  return;
end
```

⟨ ***Output procedures*** ⟩ ≡

```
def AppendBytes($src, $len):
  register $ch, $avail, $tmp;
   # Compare available space against $len rather than adding $out_end + $len,
   # which would wrap silently on a (theoretical) 64-bit overflow.
  $avail = OUT_MAX - $out_end;
  $tmp = $avail  cmpu $len;
  if $tmp is nonpos:
    call ErrHalt("plmmix-tangle: output buffer full\n");
  end
  ABLoop:
  if $len is zero:
    return;
  end
  $ch = [$src].bu;
  $OutputBuf[$out_end].bu = $ch;
  $out_end = $out_end + 1;
  $src = $src + 1;
  $len = $len - 1;
  goto ABLoop;
end
```

⟨ ***Output procedures*** ⟩ ≡

```
def AppendCStr($ptr):
  register $len;
  $len = call StrLen($ptr);
  call AppendBytes($ptr, $len);
  return;
end
```

⟨ ***Output procedures*** ⟩ ≡

```
def AppendInt($n):
   # Caller must guarantee $n >= 0.  Negative values are treated as huge positives
   # by the unsigned division, producing garbage output.
  register $count, $digit, $tmp, $ptr;


  if $n is zero:
    call AppendByte('0');
    return;
  end
  $count = 0;
  AILoop:
  goto AIEmit if $n is zero;
  $tmp = $n / 10;
  $digit = $tmp * 10;
  $digit = $n - $digit;
  $digit = $digit + '0';
  $ptr = $IntBuf + $count;
  [$ptr].bu = $digit;
  $count = $count + 1;
  $n = $tmp;
  goto AILoop;
  AIEmit:
  if $count is zero:
    return;
  end
  $count = $count - 1;
  $ptr = $IntBuf + $count;
  $digit = [$ptr].bu;
  call AppendByte($digit);
  goto AIEmit;
end
```

⟨ ***Output procedures*** ⟩ ≡

```
def AppendQuoted($ptr, $len):
  call AppendByte(34);
  call AppendBytes($ptr, $len);
  call AppendByte(34);
  return;
end
```

⟨ ***Output procedures*** ⟩ ≡

```
def EmitLineDirective($nptr, $nlen, $fptr, $flen, $line):
  register $tmp;
  $tmp = &StrLineHash;
  call AppendCStr($tmp);
  call AppendQuoted($nptr, $nlen);
  call AppendByte(' ');
  call AppendQuoted($fptr, $flen);
  call AppendByte(' ');
  call AppendInt($line);
  call AppendByte('\n');
  return;
end
```

After expansion, `$out_end` bytes are waiting in `$OutputBuf`. One `FWRITE` call flushes them all to standard output. And we are done.

⟨ ***Write output to stdout***⟩ ≡

```
$out = $FWriteArgs;
[$FWriteArgs].o       = $OutputBuf;
[$FWriteArgs + 8].o   = $out_end;
trap 0, FWRITE, STDOUT;
```

D Bootstrap Weave: Woven Output

"Weave us a garment of brightness;
May the warp be the white light of morning,
May the weft be the red light of evening. . . "
— Traditional Tewa Pueblo Song, translated by Herbert J. Spinden in *Songs of the Tewa* (1933)

This appendix shows the complete woven output of the bootstrap weaver source `plmmix-weave.plw`.

1. 1. Introduction.

This program implements the `plmmix-weave` step for the PL/MMIX literate programming system: it reads a `.plw` source from standard input and writes a LaTeX document to standard output. The weave phase is the documentation counterpart of tangle (see the `plmmix-tangle` program in the same bootstrap directory): tangle discards all prose and produces compilable code, whereas weave renders the prose as typeset text and the code chunks as pretty-printed listings suitable for reading.

Role in the bootstrap chain

Like `plmmix-tangle`, `plmmix-weave` is built from a `.plw` source, so the first binary must be produced by an already-available compatible literate toolchain. The four build steps are:

```
plmtangle -o plmmix-weave.plm plmmix-weave.plw
plmc plmmix-weave.plm -o plmmix-weave.mms
mmixal -o plmmix-weave.o plmmix-weave.mms
mmix -fsource.plw plmmix-weave.o source.plw > source.tex
```

The literate format, briefly

A `.plw` file interleaves LaTeX prose sections with PL/MMIX code chunks. Four active sigils (all beginning with an at-sign) control the structure; all other bytes are literal content. Because at-signs are active everywhere in `.plw` source, including inside LaTeX prose and inside code comments, a literal at-sign is written as `@@`. The examples below use that doubled-at escape; their exact recognition rules appear in the parsing section.

`@*` Begins a major section; the heading text follows on the same line and is consumed by the parser. The weave phase assigns section numbers automatically in source order, but the heading text itself is rendered literally.

`@` followed by whitespace Begins a minor prose section (no heading).

`@<`*name*`@>=` Defines (or appends to) a named chunk; the body follows until the next sigil.

`@<`*name*`@>` Inside code, expands a named chunk inline at that position.

`@@` A literal at-sign character in code or prose.

What weave produces

The output is a self-contained LaTeX document with the following fixed shape:

- A preamble using the article class plus `fontenc`, `inputenc`, `geometry`, `fancyvrb`, `hyperref`, `microtype`, and `booktabs`, together with a small set of macros analogous to those in `cwebmac.tex`.

- A title page and a table of contents.
- One numbered section per major prose section, in source order.
- Each code chunk typeset as a `PLMCode` fancyvrb block, with keywords bold, register names in roman italic, comments in italic, and chunk references as hyperlinks back to their definitions.
- A final cross-reference index listing every chunk name, every label definition, and every numeric-register use, each with the section numbers in which it appears.

Like `plmmix-tangle`, this is a bootstrap tool meant to run on bare MMIX without dynamic allocation. All major tables therefore have fixed compile-time limits, collected in the next section.

2. Constants and limits.

All buffer sizes are fixed at compile time. If any limit is exceeded the program halts immediately with a message on standard error.

The values below are sized for the current PL/MMIX manual and companion examples while leaving comfortable headroom for larger literate sources.

⟨ ***Constants*** ⟩ ≡

```
const SOURCE_MAX      = 1048576;    # 1 MB: max source file size
const SOURCE_NAME_MAX = 1024;       # max source filename length
const NAME_BUF_SIZE   = 131072;     # 128 KB: normalized chunk-name storage
const SECTION_MAX     = 8192;       # max parsed sections
const UNIQUE_MAX      = 2048;       # max unique chunk names
const LABEL_MAX       = 4096;       # max unique label definitions
const PROC_MAX        = 1024;       # max unique procedure definitions
const CONST_MAX       = 2048;       # max unique constant definitions
const REGISTER_MAX    = 256;        # max numeric registers
const SEC_REF_MAX     = 32768;      # max section references in unique tables
const OUT_MAX         = 4194304;    # 4 MB: output buffer
const SECTION_NEEDSPACE_LINES = 6;  # reserve a section heading plus opening prose lines
const CHUNK_NEEDSPACE_LINES = 12;   # reserve heading plus enough body to avoid orphaned
 chunk titles
const CHUNK_HEADKEEP_LINES  = 4;    # for long chunks, keep the heading with at least these
 first code lines
const CHUNK_SAMEPAGE_MAX    = 28;   # short/medium chunks stay on one page
```

Each parsed section is recorded in a fixed-size entry. A section is either a prose section (possibly with a heading) or a code chunk definition. We pack six 64-bit fields into 48 bytes per section.

⟨ ***Constants*** ⟩ ≡

```
const SEC_ENTRY_SIZE = 48;
const SEC_TABLE_SIZE = SECTION_MAX * SEC_ENTRY_SIZE;
# Section entry fields (byte offsets within entry):
const SEC_KIND  = 0;     # 0 = prose, 1 = code chunk
const SEC_HPTR  = 8;     # heading/name pointer into SourceBuf (or NameBuf for code)
const SEC_HLEN  = 16;    # heading/name length
const SEC_TPTR  = 24;    # text/body pointer into SourceBuf
const SEC_TLEN  = 32;    # text/body length
const SEC_NUM   = 40;    # prose: section number; code: definition ordinal
```

Each unique chunk entry holds a pointer and length into NameBuf plus two slots into a packed section-reference pool (one for definitions, one for uses). Label and constant entries are simpler: a single definition section plus a list of uses. Register entries hold just the register number and a list of section references.

⟨ ***Constants*** ⟩ ≡

```
const UNIQ_ENTRY_SIZE = 48;
const UNIQ_NPTR = 0;      # name pointer into NameBuf
const UNIQ_NLEN = 8;      # name length
const UNIQ_DEFS = 16;     # offset into SecRefPool for def list


const UNIQ_DLEN = 24;     # number of def entries
const UNIQ_USES = 32;     # offset into SecRefPool for use list
const UNIQ_ULEN = 40;     # number of use entries

const LBL_ENTRY_SIZE = 40;
const LBL_NPTR = 0;       # pointer into NameBuf
const LBL_NLEN = 8;
const LBL_DEF  = 16;      # defining section number
const LBL_USES = 24;      # offset into SecRefPool
const LBL_ULEN = 32;      # number of use entries

const CONST_ENTRY_SIZE = 40;
const CONST_NPTR = 0;     # pointer into NameBuf
const CONST_NLEN = 8;
const CONST_DEF  = 16;    # defining section number
const CONST_USES = 24;    # offset into SecRefPool
const CONST_ULEN = 32;    # number of use entries

const PROC_ENTRY_SIZE = 24;
const PROC_NPTR = 0;      # pointer into SourceBuf
const PROC_NLEN = 8;      # procedure-name length
const PROC_DEF  = 16;     # defining section number

const REG_ENTRY_SIZE = 24;
const REG_NUM  = 0;       # register number (numeric only)
const REG_USES = 8;       # offset into SecRefPool
const REG_ULEN = 16;
```

The scratch reference-name buffer size. Chunk-reference names longer than this overflow and halt.

⟨ ***Constants*** ⟩ ≡

```
const MAX_REF_SIZE = 512;
```

Hash-table sizes for accelerating name lookup in the four unique-key tables. Each hash table is a flat array of octa slots holding $idx + 1$ (zero meaning empty), addressed by linear probing. Sizes are powers of two so the modulo reduces to a bitwise AND, and each is at least $2\times$ the corresponding entry-count cap to keep the load factor below 0.5. A scratch key buffer slightly larger than the largest unique-table entry (currently 48 bytes) holds the pivot record during shift-based insertion sort.

⟨ ***Constants*** ⟩ ≡

```
const CHUNK_HASH_SIZE = 4096;      # >= 2 * UNIQUE_MAX
const CHUNK_HASH_MASK = 4095;
const LABEL_HASH_SIZE = 8192;      # >= 2 * LABEL_MAX
const LABEL_HASH_MASK = 8191;
const CONST_HASH_SIZE = 4096;      # >= 2 * CONST_MAX
const CONST_HASH_MASK = 4095;
const PROC_HASH_SIZE  = 2048;      # >= 2 * PROC_MAX
const PROC_HASH_MASK  = 2047;
const SORT_KEY_BUF_SIZE = 64;      # >= max UNIQ_ENTRY_SIZE/LBL_ENTRY_SIZE/etc.
const KW_HASH_SIZE = 256;          # >= ~4x PL/MMIX keyword count for low load
const KW_HASH_MASK = 255;
# Character-class bit assignments (see |$CharClass| in the data segment).
const CC_DIGIT     = #01;
const CC_ALPHA_US  = #02;
const CC_HEX_ALPHA = #04;
const CC_SPACE32   = #08;
const CC_ALNUM_US  = #03;           # digit | alpha_us
const CC_HEX       = #05;           # digit | hex_alpha
```

3. Data segment.

Everything lives in the data segment. The layout follows the processing pipeline: raw source enters SourceBuf, normalized chunk names accumulate in NameBuf, the parser builds the section table, the unique-table builder fills four flat tables (chunks, labels, constants, registers), and the LaTeX renderer assembles OutputBuf.

SourceBuf is written once by ReadAll and thereafter treated as immutable. All prose and code-body pointers reference slices of it directly; no body text is ever copied. NameBuf is append-only. OutputBuf is assembled entirely before any output is written; a single FWRITE call at the end of main flushes it.

⟨ ***Data segment*** ⟩ ≡

```
segment  data $DataBase;
# MMIX FREAD/FWRITE use two-octa argument blocks: buffer pointer, byte count.
greg $FReadArgs[16];
greg $FWriteArgs[16];


# Character classification table (one byte per ASCII value).  Bits:
#   #01 = ASCII digit,  #02 = alpha or underscore,
#   #04 = hex letter (A-F, a-f),  #08 = whitespace (<=32).
# Filled once at startup by |InitCharClass|.  Declared here, at the
# very top of the data segment, so it sits at an octa-aligned
# address and is comfortably distant from |$RefNameBuf|, which is
# known to overflow on malformed chunk-reference sequences in prose.
greg $CharClass[256];
# Keyword hash table.  Each slot holds an offset (into |Keywords|)
# of the keyword's length byte, biased by 1 so that 0 means empty.
# Filled at startup by |InitKwHash|.  Lives next to |$CharClass|,
# in the safe top-of-segment region, away from |$RefNameBuf|.
greg $KwHashTable[KW_HASH_SIZE * 8];
# Buffers
greg $SourceBuf[SOURCE_MAX];
greg $SourceNameBuf[SOURCE_NAME_MAX];
greg $NameBuf[NAME_BUF_SIZE];
greg $SectionTable[SEC_TABLE_SIZE];
```

```
greg $UniqueChunks[UNIQUE_MAX * UNIQ_ENTRY_SIZE];
greg $LabelTable[LABEL_MAX * LBL_ENTRY_SIZE];
greg $ConstantTable[CONST_MAX * CONST_ENTRY_SIZE];
greg $ProcedureTable[PROC_MAX * PROC_ENTRY_SIZE];
greg $RegisterTable[REGISTER_MAX * REG_ENTRY_SIZE];
greg $SecRefPool[SEC_REF_MAX * 8];    # packed 8-byte section numbers
# Hash tables keyed by name (FNV-1a-64).  Each slot is an octa storing
# $idx+1$ into the corresponding flat unique-key table; zero means empty.
greg $ChunkHashTable[CHUNK_HASH_SIZE * 8];
greg $LabelHashTable[LABEL_HASH_SIZE * 8];
greg $ConstHashTable[CONST_HASH_SIZE * 8];
greg $ProcHashTable [PROC_HASH_SIZE  * 8];
# Scratch pivot buffer for shift-based insertion sort.
greg $SortKeyBuf[SORT_KEY_BUF_SIZE];
greg $OutputBuf[OUT_MAX];
greg $IntBuf[32];
greg $RefNameBuf[MAX_REF_SIZE];
# Global counters (all start at 0)
greg $src_len  = 0;
greg $name_end = 0;
greg $sec_cnt  = 0;
greg $uc_cnt   = 0;
greg $lbl_cnt  = 0;
greg $const_cnt = 0;
greg $proc_cnt = 0;
greg $reg_cnt  = 0;
greg $ref_pool_end = 0;
greg $out_end  = 0;
greg $total_sections = 0;
```

The default source name and the error messages. At-at escapes are used inside string literals wherever a literal at-sign is needed, because the weave parser is active inside byte declarations as well.

⟨ ***Data segment*** ⟩ ≡

```
byte DefaultSourceName "stdin.plw", 0;
byte KwConst         "const", 0;
```

The keyword table is stored as a packed sequence of length-prefixed byte strings, terminated by a zero length byte. IsPLMKeyword walks this table linearly. The list matches the PL/MMIX keyword set used by the bootstrap tools.

⟨ ***Data segment*** ⟩ ≡

```
byte Keywords
 2, "if",     4, "else",   3, "end",    5, "while",  2, "do",
 4, "goto",   4, "call",   6, "return",
 3, "def",
 6, "likely", 8, "unlikely",
 2, "is",     6, "nonneg", 6, "nonpos", 7, "nonzero",
 3, "neg",    3, "pos",    4, "zero",   3, "odd",    4, "even",
 7, "nearest", 2, "up",    4, "down",
 7, "segment", 4, "data",  4, "code",   4, "pool",
 4, "byte",    4, "wyde",  5, "tetra",  4, "octa",
 4, "greg",    7, "reserve", 3, "loc",
 8, "register", 5, "const",  6, "prefix", 5, "local", 6, "extern",
 4, "trap",  4, "trip",  6, "resume", 8, "rounding", 4, "save", 6, "unsave",
```

```
4, "sync", 5, "ldvts", 5, "preld", 5, "prego", 5, "prest", 5, "syncd", 6, "syncid",
4, "andn", 3, "orn",  4, "nand", 3, "nor", 4, "nxor", 3, "mor", 4, "mxor",
3, "mux",  4, "sadd", 4, "bdif", 4, "wdif", 4, "tdif", 4, "odif",
3, "cmp",  4, "cmpu", 5, "cswap",
0;
```

The LaTeX preamble is stored as one long null-terminated byte string. Keeping it in one place makes the rendered document style easy to audit and update.

⟨ ***Data segment*** ⟩ ≡

```
byte TexPreamble
 "\\documentclass[10pt]{article}\n",
 "\\usepackage[T1]{fontenc}\n",
 "\\usepackage[utf8]{inputenc}\n",


 "\\usepackage{geometry}\n",
 "\\usepackage{fancyvrb}\n",
 "\\usepackage{fvextra}\n",
 "\\usepackage{enumitem}\n",
 "\\usepackage{needspace}\n",
 "\\usepackage[colorlinks,linkcolor=blue,urlcolor=blue,citecolor=blue]{hyperref}\n",
 "\\usepackage[expansion=false]{microtype}\n",
 "\\usepackage{booktabs}\n",
 "\\geometry{margin=2.5cm}\n",
 "\n",
 "\\makeatletter\\newcommand{\\plmdefanchor}[1]{\\Hy@raisedlink{\\hypertarget{#1}{}}}\\make⌋
   atother\n",
 "\\newcommand{\\plmchunkname}[1]{$\\langle$\\textit{#1}$\\rangle$}\n",
 "\\newcommand{\\plmref}[2]{\\hyperlink{#1}{\\plmchunkname{#2}}}\n",
 "\\newcommand{\\yskip}{\\smallskip}\n",
 "\\DefineVerbatimEnvironment{PLMCode}{Verbatim}{",
 "commandchars=\\\\\\{\\},fontsize=\\footnotesize,xleftmargin=1em,breaklines=true,breakanyw⌋
   here=true,breaksymbolleft={},breakindent=1em,baselinestretch=1.05}\n",
 "\n", 0;

byte StrTitleOpen    "\\title{\\texttt{", 0;
byte StrTitleClose
 "}}\n\\date{}\n\\begin{document}\n\\maketitle\n\\tableofcontents\n\\bigskip\n\n", 0;
byte StrEndDoc       "\\end{document}\n", 0;
byte StrChunkAnchor  "plmchunk.", 0;
byte StrLabelAnchor  "plmlabel.", 0;
byte StrConstAnchor  "plmconst.", 0;
byte StrProcAnchor   "plmproc.", 0;
byte StrSecAnchor    "plmsec.", 0;
byte KwByte          "byte", 0;
byte KwWyde          "wyde", 0;
byte KwOcta          "octa", 0;
byte KwTetra         "tetra", 0;
byte KwReserve       "reserve", 0;
byte StrHyperStart   "\\hypertarget{", 0;
byte StrHyperMid     "}{}", 0;
byte StrRaisedHyperStart "\\plmdefanchor{", 0;
byte StrRaisedHyperClose "}", 0;
byte StrSectionStar  "\\section*{", 0;
byte StrAddTocSection "\\addcontentsline{toc}{section}{", 0;
byte StrMedSkip      "\\medskip\\noindent\n", 0;
byte StrChunkDefEq   "}\\quad$\\equiv$\n", 0;
byte StrChunkContEq    "}}\\quad$+\\!\\equiv$\n", 0;
```

```
byte StrNoPageBreak  "\\nopagebreak\n", 0;
byte StrNeedSpaceOpen "\\Needspace{", 0;
byte StrNeedSpaceClose "\\baselineskip}\n", 0;
byte StrBeginSamePage "\\begin{samepage}\n", 0;
byte StrEndSamePage   "\\end{samepage}\n", 0;
byte StrBeginPLM     "\\begin{PLMCode}\n", 0;
byte StrBeginPLMSamePage "\\begin{PLMCode}[samepage=true]\n", 0;
byte StrEndPLMCont   "\\end{PLMCode}\n", 0;
byte StrEndPLM       "\\end{PLMCode}\n\n", 0;
byte StrBfChunkName  "\\textbf{\\plmchunkname{", 0;
byte StrHrule        "\\hrule\\bigskip\n", 0;
byte StrXrefHead     "\\section*{Cross-reference index}\n\n", 0;
byte StrSubChunks    "\\subsection*{Chunks}\n\n", 0;
byte StrSubLabels    "\\subsection*{Labels}\n\n", 0;
byte StrSubConstants "\\subsection*{Constants}\n\n", 0;
byte StrSubRegisters "\\subsection*{Registers}\n\n", 0;
byte StrBeginDesc    "\\begin{description}\n", 0;
byte StrEndDesc      "\\end{description}\n\n", 0;
byte StrItemOpen     "\\item[", 0;
byte StrItemClose    "]", 0;
byte StrDefinedIn    " defined in ", 0;
byte StrUsedIn       "; used in ", 0;
byte StrDashDash     "---", 0;
byte StrHyperlink    "\\hyperlink{", 0;
byte StrBraceClose   "}", 0;
byte StrBraceOpen    "{", 0;
byte StrSS           "\\S", 0;
byte StrPreamble     "(preamble)", 0;
byte StrTexttt       "\\texttt{", 0;
byte StrItshape      "{\\itshape ", 0;
byte StrBfseries     "{\\bfseries ", 0;
byte StrNormItReg    "{\\normalfont\\itshape\\$", 0;
byte StrPlmref       "\\plmref{", 0;
byte StrDefMarker    "~[def]", 0;
byte StrDotSpace     ".\\ ", 0;
byte StrCommaSpace   ", ", 0;
byte StrBackslashTb  "\\textbackslash{}", 0;
byte StrBraceLOpen   "\\{", 0;
byte StrBraceLClose  "\\}", 0;
byte StrAmp          "\\&", 0;
byte StrPercent      "\\%", 0;
byte StrHash         "\\#", 0;
byte StrUnder        "\\_", 0;
byte StrDollar       "\\$", 0;
byte StrCaret        "\\^{}", 0;
byte StrTilde        "\\textasciitilde{}", 0;
byte StrBeginVerbOpen "\\begin{verbatim}", 0;
byte StrEndVerbOpen   "\\end{verbatim}", 0;
```

4. Main program.

The top-level flow mirrors the phases described in the introduction: read the entire input into SourceBuf; parse it into the section table; walk the parsed sections to build the unique tables; sort those tables alphabetically or numerically; emit the LaTeX document; and finally flush the output.

The filename, used only in the document title, is captured from argv[1] if present; otherwise the default stdin.plw is used.

⟨ ***Program*** ⟩ ≡

```
⟨ Constants ⟩
⟨ Data segment ⟩
 segment  code;
 register $out = $255;
 call InitSourceName($0, $1);
 call InitCharClass();
 call InitKwHash();
 call ReadAll();
 call ParsePLW();
 call BuildUniqueTables();
 call SortConstants();
 call EmitDocument();
⟨ Write output to stdout ⟩
 trap 0, HALT, 0;
⟨ Procedures ⟩
```

5. Procedure inventory and tangled layout.

The procedures below are declared in load/execution order. This section is both the grep-target index for finding any routine by name and the authoritative statement of the helper order in the tangled PL/MMIX output. The narrative order that follows regroups the same procedures by topic: input first, then character predicates, string utilities, section-table helpers, parsing, unique-table construction, sorting, and finally document emission.

⟨ ***Procedures*** ⟩ ≡

```
⟨ ReadAll procedure ⟩
⟨ InitCharClass procedure ⟩
⟨ InitSourceName procedure ⟩
⟨ Character predicates ⟩
⟨ String utilities ⟩
⟨ Section table helpers ⟩
⟨ ParsePLW procedure ⟩
⟨ Hash table helpers ⟩
⟨ Unique table procedures ⟩
⟨ Sorting procedures ⟩
⟨ Output helpers ⟩
⟨ Anchor emitters ⟩
⟨ Prose rendering ⟩
⟨ Code body rendering ⟩
⟨ Document emission ⟩
⟨ Cross-reference scans ⟩
⟨ Error handler ⟩
```

6. Reading the input.

ReadAll fills SourceBuf in one FREAD call, then stores the actual byte count in *src_len*.*Thetrapcontractmatches tangle* :255 holds requested-count minus bytes-transferred, so zero on a full read; the actual count is SOURCE_MAX plus the return value. An exact fill is treated as overflow.

⟨***ReadAll procedure***⟩ ≡

```
def ReadAll():
  register $tmp;
 $tmp = $FReadArgs;
 [$tmp].o      = $SourceBuf;
 [$tmp + 8].o = SOURCE_MAX;
 $out = $tmp;
  trap 0, FREAD, STDIN;
 $src_len = $out + SOURCE_MAX;
  goto RASizeOk  if $src_len  is  nonneg;
  call ErrHalt("plmmix-weave: source too large\n");
RASizeOk:
 $tmp = SOURCE_MAX;
  if $src_len >=u $tmp:
    call ErrHalt("plmmix-weave: source too large\n");
  end
  return;
end
```

`InitCharClass` fills the 256-byte character classification table used by the byte predicates. Bytes 0..32 get the whitespace bit; ASCII digits get the digit bit; `A`–`F` and `a`–`f` get both the alpha bit (which also covers underscore) and the hex-letter bit; the remaining letters and the underscore character get the alpha bit alone. The data segment is zero-init by MMIX, so untouched entries stay zero.

⟨***InitCharClass procedure***⟩ ≡

```
def InitCharClass():
  register $i, $end, $cls;
 $i = 0;
ICCSpaceLoop:


 $end = $i - 33;
  goto ICCSpaceDone  if $end  is  nonneg;
 [$CharClass + $i].b = CC_SPACE32;
 $i = $i + 1;
  goto ICCSpaceLoop;
ICCSpaceDone:
 $i = '0';
ICCDigLoop:
 $end = $i - '9';
  goto ICCDigDone  if $end  is  pos;
 [$CharClass + $i].b = CC_DIGIT;
 $i = $i + 1;
  goto ICCDigLoop;
ICCDigDone:
 $i = 'A';
ICCUpHexLoop:
 $end = $i - 'F';
  goto ICCUpHexDone  if $end  is  pos;
 $cls = CC_ALPHA_US | CC_HEX_ALPHA;
 [$CharClass + $i].b = $cls;
 $i = $i + 1;
  goto ICCUpHexLoop;
ICCUpHexDone:
 $i = 'G';
ICCUpLoop:
 $end = $i - 'Z';
```

```
    goto ICCUpDone  if $end  is  pos;
   [$CharClass + $i].b = CC_ALPHA_US;
   $i = $i + 1;
    goto ICCUpLoop;
   ICCUpDone:
   $i = 'a';
   ICCLoHexLoop:
   $end = $i - 'f';
    goto ICCLoHexDone  if $end  is  pos;
   $cls = CC_ALPHA_US | CC_HEX_ALPHA;
   [$CharClass + $i].b = $cls;
   $i = $i + 1;
    goto ICCLoHexLoop;
   ICCLoHexDone:
   $i = 'g';
   ICCLoLoop:
   $end = $i - 'z';
    goto ICCLoDone  if $end  is  pos;
   [$CharClass + $i].b = CC_ALPHA_US;
   $i = $i + 1;
    goto ICCLoLoop;
   ICCLoDone:
   [$CharClass + '_'].b = CC_ALPHA_US;
    return;
  end
```

InitKwHash inserts every entry of the packed Keywords table into $KwHashTable using FNV-1a hashing (the same NameHash used by the chunk/label/proc tables) with linear probing. Each occupied slot stores the offset of the keyword's length byte (relative to Keywords), biased by 1 so the zero-initialized state means empty. IsPLMKeyword later reads a slot, subtracts 1, and recovers the matching length-prefixed keyword in one step.

⟨ ***InitCharClass procedure*** ⟩ ≡

```
def InitKwHash():
  register $tab, $base, $off, $klen, $h, $slot, $stored, $tmp;
 $base = &Keywords;
 $tab  = $base;


 IKHNext:
 $klen = [$tab].bu;
  goto IKHDone  if $klen  is  zero;
 $off = $tab - $base;
 $tmp = $tab + 1;
 $h =  call NameHash($tmp, $klen);
 $slot = $h & KW_HASH_MASK;
 IKHProbe:
 $tmp = $slot << 3;                  # * 8
 $tmp = $tmp + $KwHashTable;
 $stored = [$tmp].o;
  goto IKHInsert  if $stored  is  zero;
 $slot = $slot + 1;
 $slot = $slot & KW_HASH_MASK;
  goto IKHProbe;
 IKHInsert:
 $stored = $off + 1;
 [$tmp].o = $stored;
 $tab = $tab + $klen;
 $tab = $tab + 1;
```

```
  goto IKHNext;
 IKHDone:
  return;
end
```

InitSourceName copies argv[1] into SourceNameBuf, falling back to the default name.

⟨ ***InitSourceName procedure*** ⟩ ≡

```
def InitSourceName($argc, $argv):
  register $src, $dst, $limit, $ch, $tmp;
 $dst = $SourceNameBuf;
 $limit = SOURCE_NAME_MAX - 1;
 $src = &DefaultSourceName;
  goto ISUseDefault  if $argc < 2;
 $tmp = $argv + 8;
 $src = [$tmp].o;
  goto ISUseDefault  if $src  is  zero;
  goto ISCopy;
 ISUseDefault:
 $src = &DefaultSourceName;
 ISCopy:
 ISLoop:
  goto ISDone  if $limit  is  zero;
 $ch = [$src].bu;
  goto ISDone  if $ch  is  zero;
 [$dst].bu = $ch;
 $dst = $dst + 1;
 $src = $src + 1;
 $limit = $limit - 1;
  goto ISLoop;
 ISDone:
 [$dst].bu = 0;
  return;
end
```

7. Character predicates.

Five small predicates return 1 if their character argument falls in the indicated class, 0 otherwise. All classifications use ASCII ranges only.

The five predicates all reduce to a single `LDBU` + `AND` against `$CharClass`, filled at startup by `InitCharClass`. Each predicate masks for its bit and returns 0 or the bit value (which callers treat as a boolean via `is zero` / `is nonzero`).

⟨ ***Character predicates*** ⟩ ≡

```
def IsDigit($c) -> $r:
 $r = [$CharClass + $c].bu;
 $r = $r & CC_DIGIT;
  return $r;
end
```

⟨ ***Character predicates*** ⟩ ≡

```
def IsAlphaUS($c) -> $r:
 $r = [$CharClass + $c].bu;
 $r = $r & CC_ALPHA_US;
  return $r;
end
```

⟨ ***Character predicates*** ⟩ ≡

```
def IsAlnumUS($c) -> $r:
 $r = [$CharClass + $c].bu;
 $r = $r & CC_ALNUM_US;
  return $r;
end
```

⟨ ***Character predicates*** ⟩ ≡

```
def IsHexDigit($c) -> $r:
 $r = [$CharClass + $c].bu;
 $r = $r & CC_HEX;
  return $r;
end
```

IsSpace32 returns 1 if the byte is less than or equal to the ASCII space (32).

⟨ ***Character predicates*** ⟩ ≡

```
def IsSpace32($c) -> $r:
 $r = [$CharClass + $c].bu;
 $r = $r & CC_SPACE32;
  return $r;
end
```

8. String utilities.

The string library here is richer than the tangle tool's because the weave tool must scan and compare substrings from many contexts: detecting verbatim-environment boundaries in prose, matching keywords in code, comparing chunk names for unique-table lookup, and so on. All routines treat strings as raw byte arrays; none assume any particular encoding.

⟨***String utilities***⟩ ≡

```
def StrLen($s) -> $n:
  register $c;
 $n = 0;
 SLLoop:
 $c = [$s].bu;
  if $c  is  zero:
    return $n;
  end
 $s = $s + 1;
 $n = $n + 1;
  goto SLLoop;
end
```

StrEq compares two byte ranges of given lengths; returns 1 if equal, 0 otherwise.

⟨***String utilities***⟩ ≡

```
def StrEq($p1, $l1, $p2, $l2) -> $r:
  register $c1, $c2, $tmp;
 $tmp = $l1 - $l2;
  if $tmp  is  nonzero:
    return 0;
  end
 SELoop:
  if $l1  is  zero:
    return 1;
  end
 $c1 = [$p1].bu;
 $c2 = [$p2].bu;
 $tmp = $c1 - $c2;
  if $tmp  is  nonzero:
    return 0;
  end
 $p1 = $p1 + 1;
 $p2 = $p2 + 1;
 $l1 = $l1 - 1;
  goto SELoop;
end
```

StrCmp returns a negative, zero, or positive octa describing the lexicographic order of two byte ranges. Used by the sort routines.

⟨***String utilities***⟩ ≡

```
def StrCmp($p1, $l1, $p2, $l2) -> $r:
  register $c1, $c2, $t;
 $r = 0;
 SCLoop:
  goto SCLen  if $l1  is  zero;
  goto SCLen  if $l2  is  zero;
 $c1 = [$p1].bu;
 $c2 = [$p2].bu;
 $t = $c1 - $c2;
  if $t  is  nonzero:
    return $t;
  end
 $p1 = $p1 + 1;
```

```
    $p2 = $p2 + 1;
    $l1 = $l1 - 1;
    $l2 = $l2 - 1;
     goto SCLoop;
    SCLen:
    $t = $l1 - $l2;
     return $t;
  end
```

ContainsSliceCStr returns 1 if a C-string needle appears as a contiguous substring of a given source slice, 0 otherwise. A naive scan is adequate here since slices are typically single lines of prose.

⟨ ***String utilities*** ⟩ ≡

```
  def ContainsSliceCStr($ptr, $len, $needle) -> $r:
    register $nlen, $i, $j, $c1, $c2, $t, $p;
   $r = 0;
   $nlen =  call StrLen($needle);


    if $nlen  is  zero:
      return $r;
    end
   $i = 0;
   CSOuter:
   $t = $i + $nlen;
   $t = $t  cmpu $len;
    if $t  is  pos:
      return $r;
    end
   $j = 0;
   CSInner:
   $t = $j - $nlen;
    goto CSMatch  if $t  is  zero;
   $p = $ptr + $i;
   $p = $p + $j;
   $c1 = [$p].bu;
   $p = $needle + $j;
   $c2 = [$p].bu;
   $t = $c1 - $c2;
    goto CSNext  if $t  is  nonzero;
   $j = $j + 1;
    goto CSInner;
   CSNext:
   $i = $i + 1;
    goto CSOuter;
   CSMatch:
   $r = 1;
    return $r;
  end
```

MatchIncludeLine returns 1 if a single line consists of a top-level include directive of the form `include "path";`, permitting leading whitespace and trailing whitespace or a trailing `#` comment. The line may include its terminating newline.

⟨***String utilities***⟩ ≡

```
def MatchIncludeLine($ptr, $len) -> $matched:
  register $cur, $end, $ch, $tmp;
 $matched = 0;
 $cur = $ptr;


 $end = $ptr + $len;
MILSkipWS:
 goto MILNo  if $cur >=u $end;
$ch = [$cur].bu;
$tmp = $ch - ' ';
 goto MILStepWS  if $tmp  is  zero;
$tmp = $ch - '\t';
 goto MILStepWS  if $tmp  is  zero;
$tmp = $ch - '\n';
 goto MILNo  if $tmp  is  zero;
 goto MILKeyword;
MILStepWS:
$cur = $cur + 1;
 goto MILSkipWS;
MILKeyword:
$tmp = $end - $cur;
 goto MILNo  if $tmp < 7;
$ch = [$cur].bu;       $tmp = $ch - 'i';  goto MILNo  if $tmp  is  nonzero;
$ch = [$cur + 1].bu;   $tmp = $ch - 'n';  goto MILNo  if $tmp  is  nonzero;
$ch = [$cur + 2].bu;   $tmp = $ch - 'c';  goto MILNo  if $tmp  is  nonzero;
$ch = [$cur + 3].bu;   $tmp = $ch - 'l';  goto MILNo  if $tmp  is  nonzero;
$ch = [$cur + 4].bu;   $tmp = $ch - 'u';  goto MILNo  if $tmp  is  nonzero;
$ch = [$cur + 5].bu;   $tmp = $ch - 'd';  goto MILNo  if $tmp  is  nonzero;
$ch = [$cur + 6].bu;   $tmp = $ch - 'e';  goto MILNo  if $tmp  is  nonzero;
$cur = $cur + 7;
 goto MILNo  if $cur >=u $end;
$ch = [$cur].bu;
$tmp =  call IsSpace32($ch);
 goto MILNo  if $tmp  is  zero;
MILGap:
 goto MILNo  if $cur >=u $end;
$ch = [$cur].bu;
$tmp = $ch - ' ';
 goto MILGapStep  if $tmp  is  zero;
$tmp = $ch - '\t';
 goto MILGapStep  if $tmp  is  zero;
 goto MILQuote;
MILGapStep:
$cur = $cur + 1;
 goto MILGap;
MILQuote:
$tmp = $ch - 34;
 goto MILNo  if $tmp  is  nonzero;
$cur = $cur + 1;
MILPath:
 goto MILNo  if $cur >=u $end;
$ch = [$cur].bu;
$tmp = $ch - 34;
 goto MILAfterQuote  if $tmp  is  zero;
$tmp = $ch - '\n';
 goto MILNo  if $tmp  is  zero;
$cur = $cur + 1;
 goto MILPath;
MILAfterQuote:
$cur = $cur + 1;
```

```
MILAfterWS:
 goto MILNo  if $cur >=u $end;
$ch = [$cur].bu;
$tmp = $ch - ' ';
 goto MILAfterStep  if $tmp  is  zero;
$tmp = $ch - '\t';
 goto MILAfterStep  if $tmp  is  zero;
 goto MILSemi;
MILAfterStep:
$cur = $cur + 1;
 goto MILAfterWS;
MILSemi:
$tmp = $ch - ';';
 goto MILNo  if $tmp  is  nonzero;
$cur = $cur + 1;
MILTail:
 goto MILYes  if $cur >=u $end;
$ch = [$cur].bu;
$tmp = $ch - ' ';
 goto MILTailStep  if $tmp  is  zero;
$tmp = $ch - '\t';
 goto MILTailStep  if $tmp  is  zero;
$tmp = $ch - ' #';
 goto MILComment  if $tmp  is  zero;
$tmp = $ch - '\n';
 goto MILYes  if $tmp  is  zero;
 goto MILNo;
MILTailStep:
$cur = $cur + 1;
 goto MILTail;
MILComment:
$cur = $cur + 1;
MILCommentLoop:
 goto MILYes  if $cur >=u $end;
$ch = [$cur].bu;
$tmp = $ch - '\n';
 goto MILYes  if $tmp  is  zero;
$cur = $cur + 1;
 goto MILCommentLoop;
MILYes:
$matched = 1;
MILNo:
 return $matched;
end
```

NormNameTo copies a raw name slice into a destination buffer, collapsing whitespace runs to single spaces and trimming leading and trailing whitespace. Returns the normalized length.

⟨ ***String utilities*** ⟩ ≡

```
def NormNameTo($src, $slen, $dst) -> $nlen:
  register $i, $ch, $prev_ws, $tmp;
 $i = 0;
 $nlen = 0;


 $prev_ws = 1;
 NNLoop:
  goto NNDone  if $i >=u $slen;
 $ch = [$src].bu;
 $src = $src + 1;
 $i = $i + 1;
 $tmp = $ch  cmpu 32;
  goto NNWS  if $tmp  is  nonpos;
 $prev_ws = 0;
 [$dst].bu = $ch;
 $dst = $dst + 1;
 $nlen = $nlen + 1;
  goto NNLoop;
 NNWS:
  goto NNLoop  if $prev_ws  is  nonzero;
 $prev_ws = 1;
 [$dst].bu = 32;
 $dst = $dst + 1;
 $nlen = $nlen + 1;
  goto NNLoop;
 NNDone:
  if $nlen  is  zero:
    return $nlen;
  end
 $tmp = $dst - 1;
 $ch = [$tmp].bu;
 $tmp = $ch - 32;
  if $tmp  is  nonzero:
    return $nlen;
  end
 $nlen = $nlen - 1;
  return $nlen;
end
```

9. Section table helpers.

The parsed input is a flat array of section entries built by ParsePLW. Two access routines are enough for everyone else: AddSection appends one entry and SectionEntryAt returns the entry pointer for a 0-based index. The continuation ordinal for each chunk definition is recorded directly in *SEC_NUM* by AddUniqueChunk during the unique-table pass.

⟨***Section table helpers***⟩ ≡

```
def AddSection($kind, $hptr, $hlen, $tptr, $tlen, $num) -> $idx:
  register $entry, $tmp;
 $tmp = $sec_cnt  cmpu SECTION_MAX;
  if $tmp  is  nonneg:
    call ErrHalt("plmmix-weave: too many sections\n");
  end
 $idx = $sec_cnt;
 $entry = $sec_cnt * SEC_ENTRY_SIZE;
 $entry = $entry + $SectionTable;
 [$entry].o             = $kind;
 [$entry + SEC_HPTR].o = $hptr;
 [$entry + SEC_HLEN].o = $hlen;
 [$entry + SEC_TPTR].o = $tptr;
 [$entry + SEC_TLEN].o = $tlen;
 [$entry + SEC_NUM].o  = $num;
 $sec_cnt = $sec_cnt + 1;
  return $idx;
end
```

⟨***Section table helpers***⟩ ≡

```
def SectionEntryAt($idx) -> $entry:
 $entry = $idx * SEC_ENTRY_SIZE;
  return $entry + $SectionTable;
end
```

10. Parsing the literate source.

ParsePLW scans SourceBuf once in a two-state machine and records every prose section and every code-chunk definition in the flat section table. The states mirror those of plmmix-tangle's parser:

State 0 — prose. Text accumulates from *prose_start* to the current scan position; an at-sigil ends the run.

State 1 — code. Body runs from *body_start* to the current scan position; a new at-angle definition or a prose sigil flushes the body.

Each major heading (at-star) receives a freshly incremented section number; minor prose (at-whitespace) and code chunks get section number 0 in the table. The emitter assigns them their surrounding major section's number dynamically.

Tangling note. As in plmmix-tangle, the subchunks below share a single flat label namespace (`PPMain`, `PPAt`, `PPDone`, `PPUnclosed`, ...) and must be tangled in the order listed in `@<ParsePLW procedure@>=`. Reading one subchunk in isolation will show labels resolved by sibling subchunks; treat them as one procedure split for narrative purposes only.

⟨ ***ParsePLW procedure*** ⟩ ≡

```
def ParsePLW():
  register $scan, $send, $state;
  register $prose_start, $body_start, $body_end;
  register $cur_nptr, $cur_nlen;
  register $ch, $next, $tmp;
  register $ns, $nr, $nlen;
  register $major;
 ⟨ParsePLW: initialize scan state⟩
 ⟨ParsePLW: main scan loop⟩
 PPDone:
 ⟨ParsePLW: flush final section⟩
  return;
end
```

Initialization places the scan cursor at the start of SourceBuf and the end marker one past the last byte. The initial state is prose with an empty prose slice.

⟨ ***ParsePLW: initialize scan state*** ⟩ ≡

```
$scan = $SourceBuf;
$send = $SourceBuf + $src_len;
$state = 0;
$prose_start = $SourceBuf;
$body_start = $SourceBuf;
$body_end = $SourceBuf;
$cur_nptr = 0;
$cur_nlen = 0;
$major = 0;
```

The main scan loop advances one byte at a time. An at-sign triggers the sigil subroutine; all other bytes extend the current accumulator. In code state, every non-sigil byte extends *body_end*.

⟨ ***ParsePLW: main scan loop*** ⟩ ≡

```
PPMain:
 goto PPDone  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - '@';


 goto PPAt  if $tmp  is  zero;
 goto PPNotCode  if $state  is  zero;
$body_end = $scan + 1;
PPNotCode:
$scan = $scan + 1;
 goto PPMain;

PPAt:
$tmp = $scan + 1;
 goto PPDone  if $tmp >=u $send;
$next = [$tmp].bu;
$tmp = $next - '@';
 goto PPAtAt  if $tmp  is  zero;
$tmp = $next - '*';
 goto PPStar  if $tmp  is  zero;
$tmp = $next  cmpu 32;
 goto PPWhite  if $tmp  is  nonpos;
```

```
$tmp = $next - '<';
 goto PPAngle  if $tmp  is  zero;
$scan = $scan + 2;
 goto PPNotCodeLit  if $state  is  zero;
$body_end = $scan;
PPNotCodeLit:
 goto PPMain;

PPAtAt:
$scan = $scan + 2;
 goto PPNotCodeLit2  if $state  is  zero;
$body_end = $scan;
PPNotCodeLit2:
 goto PPMain;
```

⟨ ***ParsePLW: main scan loop*** ⟩ ≡

```
PPStar:
⟨ ParsePLW: handle at-star heading ⟩
 goto PPMain;

PPWhite:
⟨ ParsePLW: handle at-whitespace ⟩
 goto PPMain;

PPAngle:
⟨ ParsePLW: handle at-angle chunk definition ⟩
 goto PPMain;
```

An at-star sigil flushes any open chunk body or prose run, bumps the major section counter, and records a prose section whose heading slice runs from the first non-space byte after the star to just before the terminating newline. The slice is stored verbatim, including any explicit leading section number present in the source.

⟨ ***ParsePLW: handle at-star heading*** ⟩ ≡

```
 goto PPStarNoProse  if $state  is  nonzero;
$tmp = $scan - $prose_start;
 goto PPStarNoProse  if $tmp  is  zero;
 call AddSection(0, 0, 0, $prose_start, $tmp, 0);


PPStarNoProse:
 goto PPStarNoCode  if $state  is  zero;
$tmp = $body_end - $body_start;
 call AddSection(1, $cur_nptr, $cur_nlen, $body_start, $tmp, 0);
$cur_nptr = 0;
$cur_nlen = 0;
PPStarNoCode:
$state = 0;
$scan = $scan + 2;
 goto PPStarHead  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch  cmpu 32;
 goto PPStarHead  if $tmp  is  pos;
$scan = $scan + 1;
PPStarHead:
$ns = $scan;
```

```
PPStarEOL:
 goto PPStarEmit  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - '\n';
 goto PPStarEmit  if $tmp  is  zero;
$scan = $scan + 1;
 goto PPStarEOL;
PPStarEmit:
$nr = $scan - $ns;
$major = $major + 1;
 call AddSection(0, $ns, $nr, 0, 0, $major);
 goto PPStarDone  if $scan >=u $send;
$scan = $scan + 1;
PPStarDone:
$prose_start = $scan;
```

An at-whitespace sigil flushes any open chunk body and starts a new minor prose run. The prose text starts immediately after the whitespace byte.

⟨***ParsePLW: handle at-whitespace***⟩ ≡

```
 goto PPWhiteNoProse  if $state  is  nonzero;
$tmp = $scan - $prose_start;
 goto PPWhiteNoProse  if $tmp  is  zero;
 call AddSection(0, 0, 0, $prose_start, $tmp, 0);
PPWhiteNoProse:
 goto PPWhiteNoCode  if $state  is  zero;
$tmp = $body_end - $body_start;
 call AddSection(1, $cur_nptr, $cur_nlen, $body_start, $tmp, 0);
$cur_nptr = 0;
$cur_nlen = 0;
PPWhiteNoCode:
$state = 0;
$scan = $scan + 2;
$prose_start = $scan;
```

An at-angle sigil opens a chunk-name bracket. The parser scans forward for the closing at-angle, then looks for an equals sign to decide definition versus reference. For a definition, the current body or prose run is flushed and the new chunk body begins. For a reference, the sigil is simply included in the current body (if in code state) or ignored (if in prose state).

⟨***ParsePLW: handle at-angle chunk definition***⟩ ≡

```
$tmp = $scan + 2;
$ns = $tmp;
$nr = 0;
PPAngLoop:


$tmp = $ns + $nr;
 goto PPUnclosed  if $tmp >=u $send;
$ch = [$tmp].bu;
$tmp = $ch - '@';
 goto PPAngMaybe  if $tmp  is  zero;
$nr = $nr + 1;
 goto PPAngLoop;
PPAngMaybe:
$tmp = $ns + $nr;
```

```
$tmp = $tmp + 1;
 goto PPUnclosed  if $tmp >=u $send;
$ch = [$tmp].bu;
$tmp = $ch - '>';
 goto PPAngNoClose  if $tmp  is  nonzero;
 goto PPAngClose;
PPAngNoClose:
$nr = $nr + 1;
 goto PPAngLoop;
PPAngClose:
 # $nr is the raw name length.  The closing sigil sits at [$ns+$nr, $ns+$nr+2).
$tmp = $ns + $nr;
$tmp = $tmp + 2;
 goto PPAngRef  if $tmp >=u $send;
$ch = [$tmp].bu;
$tmp = $ch - '=';
 goto PPAngRef  if $tmp  is  nonzero;
 # --- Definition ---
 goto PPAngNoProse  if $state  is  nonzero;
$tmp = $scan - $prose_start;
 goto PPAngNoProse  if $tmp  is  zero;
 call AddSection(0, 0, 0, $prose_start, $tmp, 0);
PPAngNoProse:
 goto PPAngNoCode  if $state  is  zero;
$tmp = $body_end - $body_start;
 call AddSection(1, $cur_nptr, $cur_nlen, $body_start, $tmp, 0);
PPAngNoCode:
$tmp = $name_end + $nr;
$tmp = $tmp  cmpu NAME_BUF_SIZE;
 if $tmp  is  nonneg:
   call ErrHalt("plmmix-weave: name buffer overflow\n");
 end
$cur_nptr = $NameBuf + $name_end;
$cur_nlen =  call NormNameTo($ns, $nr, $cur_nptr);
$name_end = $name_end + $cur_nlen;
$scan = $ns + $nr;
$scan = $scan + 3;         # past at-angle-close plus '='
 goto PPAngBody  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - '\n';
 goto PPAngBody  if $tmp  is  nonzero;
$scan = $scan + 1;
PPAngBody:
$body_start = $scan;
$body_end = $scan;
$state = 1;
 goto PPAngDoneAll;
PPAngRef:
⟨ParsePLW: scan code chunk body⟩
PPAngDoneAll:
```

Inside code state, a chunk-reference sigil simply extends the current body so that the body slice includes the reference text verbatim. The renderer will recognize and expand those references later. In prose state, a stray at-angle is effectively ignored; the bytes following it are consumed as regular prose.

⟨ ***ParsePLW: scan code chunk body*** ⟩ ≡

```
$scan = $ns + $nr;
$scan = $scan + 2;          # past at-angle-close
 goto PPAngRefEnd  if $state  is  zero;
$body_end = $scan;
PPAngRefEnd:
 goto PPAngSkipUnclosed;
PPUnclosed:
 call ErrHalt("plmmix-weave: unclosed at-angle in source\n");
PPAngSkipUnclosed:
```

At end of input, any remaining open run is flushed as a final section.

⟨ ***ParsePLW: flush final section*** ⟩ ≡

```
 goto PPFFinCode  if $state  is  nonzero;
$tmp = $scan - $prose_start;
 goto PPFFEnd  if $tmp  is  zero;
 call AddSection(0, 0, 0, $prose_start, $tmp, 0);
 goto PPFFEnd;
PPFFinCode:
$tmp = $body_end - $body_start;
 call AddSection(1, $cur_nptr, $cur_nlen, $body_start, $tmp, 0);
PPFFEnd:
```

11. Hash tables for unique-name lookup.

The four unique-key tables (chunks, labels, constants, procedures) are looked up by name on every identifier in every code body, plus on every chunk reference and every prose inline-code fragment. A naive linear scan over each unique table dominates the cost on larger inputs, so we maintain a parallel hash table per kind keyed by the FNV-1a-64 hash of the name.

Each hash table is a flat octa array sized to a power of two so the modulo reduces to a bitwise AND. Slots store $idx + 1$ into the underlying flat unique-key table; zero means empty. Collisions are resolved by linear probing. Greg-allocated arrays start zero-filled, so no explicit initialization is required at startup.

After `SortConstants` reorders the constant table, the indices in `ConstHashTable` are stale and must be rebuilt with `HashRebuild` before any further lookup.

⟨ ***Hash table helpers*** ⟩ ≡

```
def NameHash($ptr, $len) -> $h:
  register $i, $b, $prime, $end;
  # Hoist the FNV-1a prime and the loop bound out of the body.  The
  # constant otherwise re-materialises (four SET* instructions) on
  # every byte, and the |$i >=u $len| test costs an extra subtract.
 $h = #cbf29ce484222325;
 $prime = #100000001b3;
 $end = $ptr + $len;
 NHLoop:
  if $ptr >=u $end:
    return $h;
  end
 $b = [$ptr].bu;
```

```
    $h = $h ^ $b;
    $h = $h * $prime;
    $ptr = $ptr + 1;
     goto NHLoop;
  end
```

HashFindIdx returns the entry index of a name if present, or -1 if the name is not in the table. Caller supplies the hash table base, its mask (size minus one), the underlying entry table, and the entry size; it assumes name pointer at offset 0 and length at offset 8 of each entry.

⟨ ***Hash table helpers*** ⟩ ≡

```
  def HashFindIdx($hash_tbl, $hash_mask, $table, $esize, $nptr, $nlen) -> $idx:
     register $h, $slot, $tmp, $stored, $entry_idx, $entry, $cptr, $clen, $eq;
    $idx = -1;
    $h =  call NameHash($nptr, $nlen);
    $slot = $h & $hash_mask;
    HFILoop:
    $tmp = $slot << 3;                  # * 8
    $tmp = $tmp + $hash_tbl;
    $stored = [$tmp].o;
     if $stored  is  zero:
       return $idx;
     end
    $entry_idx = $stored - 1;
    $entry = $entry_idx * $esize;
    $entry = $entry + $table;
    $cptr = [$entry].o;
    $clen = [$entry + 8].o;
    $eq =  call StrEq($nptr, $nlen, $cptr, $clen);
     goto HFINext  if $eq  is  zero;
    $idx = $entry_idx;
     return $idx;
    HFINext:
    $slot = $slot + 1;
    $slot = $slot & $hash_mask;
     goto HFILoop;
  end
```

HashInsertIdx writes $idx + 1$ into the first empty slot found by linear probe. The hash table size must exceed any possible entry count or this will loop forever; the constants in the previous section enforce that.

⟨ ***Hash table helpers*** ⟩ ≡

```
  def HashInsertIdx($hash_tbl, $hash_mask, $idx, $nptr, $nlen):
     register $h, $slot, $tmp, $cur, $val;
    $h =  call NameHash($nptr, $nlen);
    $slot = $h & $hash_mask;
    HIILoop:
    $tmp = $slot << 3;                  # * 8
    $tmp = $tmp + $hash_tbl;
    $cur = [$tmp].o;
     goto HIIPlace  if $cur  is  zero;
    $slot = $slot + 1;
    $slot = $slot & $hash_mask;
     goto HIILoop;
    HIIPlace:
```

```
  $val = $idx + 1;
  [$tmp].o = $val;
   return;
end
```

HashClear zeros every slot in a hash table. HashRebuild zeros and re-inserts every live entry of the underlying flat table; it is used after a sort reorders entries beneath us.

⟨ ***Hash table helpers*** ⟩ ≡

```
def HashClear($hash_tbl, $hash_size):
  register $i, $tmp;
 $i = 0;
 HCLoop:
  if $i >=u $hash_size:
    return;
  end
 $tmp = $i << 3;                    # * 8
 $tmp = $tmp + $hash_tbl;
 [$tmp].o = 0;
 $i = $i + 1;
  goto HCLoop;
end
```

⟨ ***Hash table helpers*** ⟩ ≡

```
def HashRebuild($hash_tbl, $hash_size, $hash_mask, $table, $count, $esize):
  register $i, $entry, $nptr, $nlen;
  call HashClear($hash_tbl, $hash_size);
 $i = 0;
 HRLoop:
  if $i >=u $count:
    return;
  end
 $entry = $i * $esize;
 $entry = $entry + $table;
 $nptr = [$entry].o;
 $nlen = [$entry + 8].o;
  call HashInsertIdx($hash_tbl, $hash_mask, $i, $nptr, $nlen);
 $i = $i + 1;
  goto HRLoop;
end
```

Per-table convenience wrappers used by the lookup-heavy callers in the unique-table builder and the renderers.

⟨ ***Hash table helpers*** ⟩ ≡

```
def FindChunkByName($nptr, $nlen) -> $idx:
 $idx =  call HashFindIdx($ChunkHashTable, CHUNK_HASH_MASK,
                          $UniqueChunks, UNIQ_ENTRY_SIZE, $nptr, $nlen);
  return $idx;
end
```

⟨ ***Hash table helpers*** ⟩ ≡

```
def FindLabelByName($nptr, $nlen) -> $idx:
 $idx =  call HashFindIdx($LabelHashTable, LABEL_HASH_MASK,
                          $LabelTable, LBL_ENTRY_SIZE, $nptr, $nlen);
  return $idx;
end
```

⟨ ***Hash table helpers*** ⟩ ≡

```
def FindConstantByName($nptr, $nlen) -> $idx:
 $idx =  call HashFindIdx($ConstHashTable, CONST_HASH_MASK,
                          $ConstantTable, CONST_ENTRY_SIZE, $nptr, $nlen);
  return $idx;
end
```

⟨ ***Hash table helpers*** ⟩ ≡

```
def FindProcedureByName($nptr, $nlen) -> $idx:
 $idx =  call HashFindIdx($ProcHashTable, PROC_HASH_MASK,
                          $ProcedureTable, PROC_ENTRY_SIZE, $nptr, $nlen);
  return $idx;
end
```

12. Unique chunk, label, constant, and register tables.

After parsing, we make one linear pass over the section table to build four unique-key tables. For each code chunk we record its name in UniqueChunks (first occurrence only) and update its defs and uses lists. For each label in each body (identifier followed by a colon) we update LabelTable. Constant definitions are recorded from `const Name = ...` declarations, then a second pass marks uses. For each numeric register in each body (a dollar followed by digits) we update RegisterTable.

Packed section-reference lists are stored in a single flat pool. Each unique entry carries a pointer (an offset into the pool) and a count. Since references are appended in section order, lists are already sorted and we only need to skip duplicates on insertion.

AppendToRefList appends a section number to a packed list. If the list has not yet been allocated (length zero) we allocate a fresh slot by assigning the current pool end. If the new section number equals the tail of the list, we skip the append.

⟨ ***Unique table procedures*** ⟩ ≡

```
def AppendToRefList($entry, $off_ptr, $off_len, $sec):
  register $base, $len, $tail, $tmp, $poolp;
 $base = [$entry + $off_ptr].o;
 $len  = [$entry + $off_len].o;


  goto ATRCheck  if $len  is  zero;
 $tail = $base + $len;
 $tail = $tail - 1;
 $poolp = $tail << 3;                # * 8
 $poolp = $poolp + $SecRefPool;
 $tmp = [$poolp].o;
 $tmp = $tmp - $sec;
  if $tmp  is  zero:
    return;
  end
  goto ATRAppend;
 ATRCheck:
 $base = $ref_pool_end;
 [$entry + $off_ptr].o = $base;
 ATRAppend:
 $tmp = $ref_pool_end  cmpu SEC_REF_MAX;
  if $tmp  is  nonneg:
    call ErrHalt("plmmix-weave: section-ref pool full\n");
  end
 $poolp = $ref_pool_end << 3;        # * 8
 $poolp = $poolp + $SecRefPool;
 [$poolp].o = $sec;
 $ref_pool_end = $ref_pool_end + 1;
 $len = $len + 1;
 [$entry + $off_len].o = $len;
  return;
end
```

AddUniqueChunk records a definition or a use. If the name is new, a new entry is allocated in UniqueChunks; the appropriate list is then extended with the current section number.

⟨ ***Unique table procedures*** ⟩ ≡

```
def AddUniqueChunk($nptr, $nlen, $sec, $is_def) -> $count:
  register $idx, $entry, $tmp;
 $count = 0;
 $idx =  call FindChunkByName($nptr, $nlen);


  goto AUCHave  if $idx  is  nonneg;
 $tmp = $uc_cnt  cmpu UNIQUE_MAX;
  if $tmp  is  nonneg:
    call ErrHalt("plmmix-weave: too many unique chunk names\n");
  end
 $idx = $uc_cnt;
 $entry = $idx * UNIQ_ENTRY_SIZE;
 $entry = $entry + $UniqueChunks;
 [$entry + UNIQ_NPTR].o = $nptr;
 [$entry + UNIQ_NLEN].o = $nlen;
 [$entry + UNIQ_DEFS].o = 0;
 [$entry + UNIQ_DLEN].o = 0;
 [$entry + UNIQ_USES].o = 0;
 [$entry + UNIQ_ULEN].o = 0;
```

```
  $uc_cnt = $uc_cnt + 1;
   call HashInsertIdx($ChunkHashTable, CHUNK_HASH_MASK, $idx, $nptr, $nlen);
   goto AUCAppend;
  AUCHave:
  $entry = $idx * UNIQ_ENTRY_SIZE;
  $entry = $entry + $UniqueChunks;
  AUCAppend:
   goto AUCUse  if $is_def  is  zero;
   call AppendToRefList($entry, UNIQ_DEFS, UNIQ_DLEN, $sec);
   return [$entry + UNIQ_DLEN].o;
  AUCUse:
   call AppendToRefList($entry, UNIQ_USES, UNIQ_ULEN, $sec);
   return $count;
end
```

AddUniqueLabel records a label definition or use. The first occurrence is treated as the definition and all subsequent occurrences as uses.

⟨ ***Unique table procedures*** ⟩ ≡

```
def AddUniqueLabel($nptr, $nlen, $sec):
  register $idx, $entry, $tmp;
 $idx =  call FindLabelByName($nptr, $nlen);
  goto ALBHave  if $idx  is  nonneg;
 $tmp = $lbl_cnt  cmpu LABEL_MAX;
  if $tmp  is  nonneg:
    call ErrHalt("plmmix-weave: too many labels\n");
  end
 $idx = $lbl_cnt;
 $entry = $idx * LBL_ENTRY_SIZE;
 $entry = $entry + $LabelTable;
 [$entry + LBL_NPTR].o = $nptr;
 [$entry + LBL_NLEN].o = $nlen;
 [$entry + LBL_DEF].o  = $sec;
 [$entry + LBL_USES].o = 0;
 [$entry + LBL_ULEN].o = 0;
 $lbl_cnt = $lbl_cnt + 1;
  call HashInsertIdx($LabelHashTable, LABEL_HASH_MASK, $idx, $nptr, $nlen);
  return;
 ALBHave:
 $entry = $idx * LBL_ENTRY_SIZE;
 $entry = $entry + $LabelTable;
  call AppendToRefList($entry, LBL_USES, LBL_ULEN, $sec);
  return;
end
```

⟨ ***Unique table procedures*** ⟩ ≡

```
def IsConstKeyword($nptr, $nlen) -> $r:
  register $ch;
 $r = 0;
 $ch = $nlen - 5;


  if $ch  is  nonzero:
    return $r;
  end
 $ch = [$nptr].bu;
 $ch = $ch - 'c';
  if $ch  is  nonzero:
    return $r;
  end
 $ch = [$nptr + 1].bu;
 $ch = $ch - 'o';
  if $ch  is  nonzero:
    return $r;
  end
 $ch = [$nptr + 2].bu;
 $ch = $ch - 'n';
  if $ch  is  nonzero:
    return $r;
  end
 $ch = [$nptr + 3].bu;
 $ch = $ch - 's';
  if $ch  is  nonzero:
    return $r;
  end
 $ch = [$nptr + 4].bu;
 $ch = $ch - 't';
  if $ch  is  nonzero:
    return $r;
  end
  return 1;
end
```

AddUniqueConstant records a constant definition or use. Definitions may arrive after uses, so the definition field is filled on the first explicit declaration and uses accumulate independently.

⟨ ***Unique table procedures*** ⟩ ≡

```
def AddUniqueConstant($nptr, $nlen, $sec, $is_def):
  register $idx, $entry, $tmp;
 $idx =  call FindConstantByName($nptr, $nlen);
  goto AUConstHave  if $idx  is  nonneg;


 $tmp = $const_cnt  cmpu CONST_MAX;
  if $tmp  is  nonneg:
    call ErrHalt("plmmix-weave: too many constants\n");
  end
 $idx = $const_cnt;
 $entry = $idx * CONST_ENTRY_SIZE;
 $entry = $entry + $ConstantTable;
 [$entry + CONST_NPTR].o = $nptr;
 [$entry + CONST_NLEN].o = $nlen;
```

```
 [$entry + CONST_DEF].o  = 0;
 [$entry + CONST_USES].o = 0;
 [$entry + CONST_ULEN].o = 0;
 $const_cnt = $const_cnt + 1;
  call HashInsertIdx($ConstHashTable, CONST_HASH_MASK, $idx, $nptr, $nlen);
  goto AUConstReady;
 AUConstHave:
 $entry = $idx * CONST_ENTRY_SIZE;
 $entry = $entry + $ConstantTable;
 AUConstReady:
  goto AUConstUse  if $is_def  is  zero;
 $tmp = [$entry + CONST_DEF].o;
  if $tmp  is  nonzero:
    return;
  end
 [$entry + CONST_DEF].o = $sec;
  return;
 AUConstUse:
  call AppendToRefList($entry, CONST_USES, CONST_ULEN, $sec);
  return;
end
```

AddProcedure records one procedure definition by name and defining section. Later duplicate sightings are ignored.

⟨ ***Unique table procedures*** ⟩ ≡

```
def AddProcedure($nptr, $nlen, $sec):
  register $idx, $entry, $tmp;
 $idx =  call FindProcedureByName($nptr, $nlen);
  if $idx  is  nonneg:
    return;
  end
 $idx = $proc_cnt;
 $tmp = $proc_cnt  cmpu PROC_MAX;
  if $tmp  is  nonneg:
    call ErrHalt("plmmix-weave: too many procedures\n");
  end
 $entry = $proc_cnt * PROC_ENTRY_SIZE;
 $entry = $entry + $ProcedureTable;
 [$entry + PROC_NPTR].o = $nptr;
 [$entry + PROC_NLEN].o = $nlen;
 [$entry + PROC_DEF].o  = $sec;
 $proc_cnt = $proc_cnt + 1;
  call HashInsertIdx($ProcHashTable, PROC_HASH_MASK, $idx, $nptr, $nlen);
  return;
end
```

MarkRegister records a numeric register use. Unlike chunks and labels, registers are keyed by an integer, so we sweep the table with a linear scan rather than a name match.

⟨ ***Unique table procedures*** ⟩ ≡

```
def MarkRegister($num, $sec):
  register $i, $entry, $cur, $tmp;
 $i = 0;
 MRLoop:
  goto MRNew  if $i >=u $reg_cnt;
 $entry = $i * REG_ENTRY_SIZE;
 $entry = $entry + $RegisterTable;
 $cur = [$entry + REG_NUM].o;
 $tmp = $cur - $num;
  goto MRHave  if $tmp  is  zero;
 $i = $i + 1;
  goto MRLoop;
 MRNew:
 $tmp = $reg_cnt  cmpu REGISTER_MAX;
  if $tmp  is  nonneg:
    call ErrHalt("plmmix-weave: too many registers\n");
  end
 $entry = $reg_cnt * REG_ENTRY_SIZE;
 $entry = $entry + $RegisterTable;
 [$entry + REG_NUM].o  = $num;
 [$entry + REG_USES].o = 0;
 [$entry + REG_ULEN].o = 0;
 $reg_cnt = $reg_cnt + 1;
 MRHave:
  call AppendToRefList($entry, REG_USES, REG_ULEN, $sec);
  return;
end
```

ScanBodyForLabelsRegs sweeps one code body and, for each label definition or numeric-register occurrence, calls AddUniqueLabel or MarkRegister. Labels are identifiers immediately followed by a colon; registers are a dollar sign followed by at least one digit.

⟨ ***Unique table procedures*** ⟩ ≡

```
def IsTokenAtLineStart($bptr, $tok_start) -> $r:
  register $p, $ch, $t;
 $r = 1;
 $p = $tok_start;
 ITALSLoop:
  if $p <=u $bptr:
    return $r;
  end
 $p = $p - 1;
 $ch = [$p].bu;
 $t = $ch - '\n';
  if $t  is  zero:
    return $r;
  end
 $t = $ch  cmpu 32;
  goto ITALSLoop  if $t  is  nonpos;
 $r = 0;
  return $r;
end
```

⟨ ***Unique table procedures*** ⟩ ≡

```
def IsDataDeclKeyword($nptr, $nlen) -> $r:
  register $ch;
 $r = 0;
 $ch = $nlen - 4;


  goto IDDKLen5  if $ch  is  nonzero;
 $ch = [$nptr].bu;
 $ch = $ch - 'b';
  goto IDDKByte  if $ch  is  zero;
 $ch = [$nptr].bu;
 $ch = $ch - 'o';
  goto IDDKOcta  if $ch  is  zero;
 $ch = [$nptr].bu;
 $ch = $ch - 'w';
  goto IDDKWyde  if $ch  is  zero;
  return $r;
 IDDKByte:
 $ch = [$nptr + 1].bu;
 $ch = $ch - 'y';
  goto IDDKNo  if $ch  is  nonzero;
 $ch = [$nptr + 2].bu;
 $ch = $ch - 't';
  goto IDDKNo  if $ch  is  nonzero;
 $ch = [$nptr + 3].bu;
 $ch = $ch - 'e';
  goto IDDKNo  if $ch  is  nonzero;
  return 1;
 IDDKOcta:
 $ch = [$nptr + 1].bu;
 $ch = $ch - 'c';
  goto IDDKNo  if $ch  is  nonzero;
 $ch = [$nptr + 2].bu;
 $ch = $ch - 't';
  goto IDDKNo  if $ch  is  nonzero;
 $ch = [$nptr + 3].bu;
 $ch = $ch - 'a';
  goto IDDKNo  if $ch  is  nonzero;
  return 1;
 IDDKWyde:
 $ch = [$nptr + 1].bu;
 $ch = $ch - 'y';
  goto IDDKNo  if $ch  is  nonzero;
 $ch = [$nptr + 2].bu;
 $ch = $ch - 'd';
  goto IDDKNo  if $ch  is  nonzero;
 $ch = [$nptr + 3].bu;
 $ch = $ch - 'e';
  goto IDDKNo  if $ch  is  nonzero;
  return 1;
 IDDKLen5:
 $ch = $nlen - 5;
  goto IDDKLen7  if $ch  is  nonzero;
 $ch = [$nptr].bu;
 $ch = $ch - 't';
  goto IDDKNo  if $ch  is  nonzero;
 $ch = [$nptr + 1].bu;
 $ch = $ch - 'e';
  goto IDDKNo  if $ch  is  nonzero;
 $ch = [$nptr + 2].bu;
 $ch = $ch - 't';
```

```
    goto IDDKNo if $ch is nonzero;
  $ch = [$nptr + 3].bu;
  $ch = $ch - 'r';
    goto IDDKNo if $ch is nonzero;
  $ch = [$nptr + 4].bu;
  $ch = $ch - 'a';
    goto IDDKNo if $ch is nonzero;
    return 1;
  IDDKLen7:
  $ch = $nlen - 7;
    goto IDDKNo if $ch is nonzero;
  $ch = [$nptr].bu;
  $ch = $ch - 'r';
    goto IDDKNo if $ch is nonzero;
  $ch = [$nptr + 1].bu;
  $ch = $ch - 'e';
    goto IDDKNo if $ch is nonzero;
  $ch = [$nptr + 2].bu;
  $ch = $ch - 's';
    goto IDDKNo if $ch is nonzero;
  $ch = [$nptr + 3].bu;
  $ch = $ch - 'e';
    goto IDDKNo if $ch is nonzero;
  $ch = [$nptr + 4].bu;
  $ch = $ch - 'r';
    goto IDDKNo if $ch is nonzero;
  $ch = [$nptr + 5].bu;
  $ch = $ch - 'v';
    goto IDDKNo if $ch is nonzero;
  $ch = [$nptr + 6].bu;
  $ch = $ch - 'e';
    goto IDDKNo if $ch is nonzero;
    return 1;
  IDDKNo:
    return $r;
  end
```

⟨ ***Unique table procedures*** ⟩ ≡

```
  def IsConstDefSite($bptr, $tok_start) -> $r:
    register $line_start, $p, $kw_start, $kw_len, $ch, $t, $alpha;
  $r = 0;
  $line_start = $tok_start;


  ICDSLineBack:
    goto ICDSLineReady if $line_start <=u $bptr;
  $p = $line_start - 1;
  $ch = [$p].bu;
  $t = $ch - '\n';
    goto ICDSLineReady if $t is zero;
  $line_start = $p;
    goto ICDSLineBack;
  ICDSLineReady:
  $p = $line_start;
  ICDSSkipLead:
    if $p >=u $tok_start:
      return;
    end
  $ch = [$p].bu;
```

```
    $t = $ch  cmpu 32;
     goto ICDSHaveKw  if $t  is  pos;
    $p = $p + 1;
     goto ICDSSkipLead;
    ICDSHaveKw:
    $alpha =  call IsAlphaUS($ch);
     if $alpha  is  zero:
       return;
     end
    $kw_start = $p;
    ICDSKwLoop:
    $p = $p + 1;
     goto ICDSKwDone  if $p >=u $tok_start;
    $ch = [$p].bu;
    $alpha =  call IsAlnumUS($ch);
     goto ICDSKwDone  if $alpha  is  zero;
     goto ICDSKwLoop;
    ICDSKwDone:
    $kw_len = $p - $kw_start;
    $t =  call IsConstKeyword($kw_start, $kw_len);
     goto ICDSDone  if $t  is  zero;
    ICDSSkipGap:
     goto ICDSDone  if $p >u $tok_start;
     goto ICDSYes  if $p >=u $tok_start;
    $ch = [$p].bu;
    $t = $ch  cmpu 32;
     goto ICDSDone  if $t  is  pos;
    $p = $p + 1;
     goto ICDSSkipGap;
    ICDSYes:
    $r = 1;
    ICDSDone:
     return;
  end
```

⟨ ***Unique table procedures*** ⟩ ≡

```
  def IsDataLabelDefSite($bptr, $tok_start) -> $r:
    register $line_start, $p, $kw_start, $kw_len, $ch, $t, $alpha;
   $r = 0;
   $line_start = $tok_start;


   IDLDLineBack:
    goto IDLDLineReady  if $line_start <=u $bptr;
   $p = $line_start - 1;
   $ch = [$p].bu;
   $t = $ch - '\n';
    goto IDLDLineReady  if $t  is  zero;
   $line_start = $p;
    goto IDLDLineBack;
   IDLDLineReady:
   $p = $line_start;
   IDLDSkipLead:
    if $p >=u $tok_start:
      return;
    end
   $ch = [$p].bu;
   $t = $ch  cmpu 32;
    goto IDLDHaveKw  if $t  is  pos;
```

```
  $p = $p + 1;
   goto IDLDSkipLead;
  IDLDHaveKw:
  $alpha =  call IsAlphaUS($ch);
   if $alpha  is  zero:
     return;
   end
  $kw_start = $p;
  IDLDKwLoop:
  $p = $p + 1;
   goto IDLDKwDone  if $p >=u $tok_start;
  $ch = [$p].bu;
  $alpha =  call IsAlnumUS($ch);
   goto IDLDKwDone  if $alpha  is  zero;
   goto IDLDKwLoop;
  IDLDKwDone:
  $kw_len = $p - $kw_start;
  $t =  call IsDataDeclKeyword($kw_start, $kw_len);
   goto IDLDDone  if $t  is  zero;
  IDLDSkipGap:
   goto IDLDDone  if $p >u $tok_start;
   goto IDLDYes  if $p >=u $tok_start;
  $ch = [$p].bu;
  $t = $ch  cmpu 32;
   goto IDLDDone  if $t  is  pos;
  $p = $p + 1;
   goto IDLDSkipGap;
  IDLDYes:
  $r = 1;
  IDLDDone:
   return;
 end
```

⟨ ***Unique table procedures*** ⟩ ≡

```
def ScanBodyForLabelsRegs($bptr, $blen, $sec):
  register $p, $end, $ch, $t, $start, $alpha, $digit, $num, $nlen, $decl_kw;
 $p = $bptr;
 $end = $bptr + $blen;


 $decl_kw = 0;
 SBLoop:
  goto SBDone  if $p >=u $end;
 $ch = [$p].bu;
 $t = $ch - '\n';
  goto SBLineBreak  if $t  is  zero;
 $t = $ch - ';';
  goto SBSemi  if $t  is  zero;
 $t = $ch - 34;
  goto SBStr  if $t  is  zero;
 $t = $ch - ' #';
  goto SBHash  if $t  is  zero;
 $t = $ch - '$';
  goto SBReg  if $t  is  zero;
 $alpha =  call IsAlphaUS($ch);
  goto SBIdent  if $alpha  is  nonzero;
 $p = $p + 1;
  goto SBLoop;
 SBLineBreak:
```

```
$decl_kw = 0;
$p = $p + 1;
 goto SBLoop;
SBSemi:
$decl_kw = 0;
$p = $p + 1;
 goto SBLoop;
SBStr:
$p = $p + 1;
SBStrLoop:
 goto SBLoop  if $p >=u $end;
$ch = [$p].bu;
$t = $ch - 92;
 goto SBStrBS  if $t  is  zero;
$t = $ch - 34;
 goto SBStrEnd  if $t  is  zero;
$p = $p + 1;
 goto SBStrLoop;
SBStrBS:
$p = $p + 2;
 goto SBStrLoop;
SBStrEnd:
$p = $p + 1;
 goto SBLoop;
SBHash:
$t = $p + 1;
 goto SBCmt  if $t >=u $end;
$ch = [$t].bu;
$digit =  call IsHexDigit($ch);
 goto SBHex  if $digit  is  nonzero;
SBCmt:
$p = $p + 1;
SBCmtLoop:
 goto SBLoop  if $p >=u $end;
$ch = [$p].bu;
$t = $ch - '\n';
 goto SBCmtEnd  if $t  is  zero;
$p = $p + 1;
 goto SBCmtLoop;
SBCmtEnd:
$decl_kw = 0;
$p = $p + 1;
 goto SBLoop;
SBHex:
$p = $p + 1;
SBHexLoop:
 goto SBLoop  if $p >=u $end;
$ch = [$p].bu;
$digit =  call IsHexDigit($ch);
 goto SBLoop  if $digit  is  zero;
$p = $p + 1;
 goto SBHexLoop;
SBIdent:
$start = $p;
SBIdentRun:
$p = $p + 1;
 goto SBIdentEnd  if $p >=u $end;
$ch = [$p].bu;
$alpha =  call IsAlnumUS($ch);
 goto SBIdentRun  if $alpha  is  nonzero;
SBIdentEnd:
 goto SBLoop  if $p >=u $end;
```

```
$ch = [$p].bu;
$t = $ch - ':';
 goto SBMaybeDataLabel  if $t  is  nonzero;
$nlen = $p - $start;
$t =  call IsTokenAtLineStart($bptr, $start);
 goto SBNotCodeLabel  if $t  is  zero;
 call AddUniqueLabel($start, $nlen, $sec);
$decl_kw = 0;
SBNotCodeLabel:
$p = $p + 1;
 goto SBLoop;
SBMaybeDataLabel:
$nlen = $p - $start;
$t =  call IsDataDeclKeyword($start, $nlen);
 goto SBMaybePendingDecl  if $t  is  zero;
$t = $p;
SBDeclSkip:
 goto SBLoop  if $t >=u $end;
$ch = [$t].bu;
$alpha =  call IsAlphaUS($ch);
 goto SBDeclName  if $alpha  is  nonzero;
$digit = $ch  cmpu 32;
 goto SBLoop  if $digit  is  pos;
$t = $t + 1;
 goto SBDeclSkip;
SBDeclName:
$start = $t;
SBDeclNameLoop:
$t = $t + 1;
 goto SBDeclNameDone  if $t >=u $end;
$ch = [$t].bu;
$alpha =  call IsAlnumUS($ch);
 goto SBDeclNameDone  if $alpha  is  zero;
 goto SBDeclNameLoop;
SBDeclNameDone:
$nlen = $t - $start;
 call AddUniqueLabel($start, $nlen, $sec);
$p = $t;
 goto SBLoop;
SBMaybePendingDecl:
 goto SBCheckDeclKeyword  if $decl_kw  is  zero;
 call AddUniqueLabel($start, $nlen, $sec);
$decl_kw = 0;
 goto SBLoop;
SBCheckDeclKeyword:
$t =  call IsDataDeclKeyword($start, $nlen);
 goto SBLoop  if $t  is  zero;
$decl_kw = 1;
 goto SBLoop;
SBReg:
$p = $p + 1;
 goto SBLoop  if $p >=u $end;
$ch = [$p].bu;
$digit =  call IsDigit($ch);
 goto SBLoop  if $digit  is  zero;
$num = 0;
SBRegLoop:
$num = $num * 10;
$t = $ch - '0';
$num = $num + $t;
$p = $p + 1;
 goto SBRegEnd  if $p >=u $end;
```

```
    $ch = [$p].bu;
    $digit =  call IsDigit($ch);
     goto SBRegLoop  if $digit  is  nonzero;
    SBRegEnd:
     call MarkRegister($num, $sec);
     goto SBLoop;
    SBDone:
     return;
   end
```

ScanBodyForProcedureDefs finds procedure definitions of the form `def Name(...` while skipping strings and line comments.

⟨ ***Unique table procedures*** ⟩ ≡

```
def ScanBodyForProcedureDefs($bptr, $blen, $sec):
  register $p, $end, $ch, $t, $alpha, $digit, $start, $nptr, $nlen, $scan;
 $p = $bptr;
 $end = $bptr + $blen;


 SBPDLoop:
  goto SBPDDone  if $p >=u $end;
 $ch = [$p].bu;
 $t = $ch - 34;
  goto SBPDStr  if $t  is  zero;
 $t = $ch - ' #';
  goto SBPDHash  if $t  is  zero;
 $alpha =  call IsAlphaUS($ch);
  goto SBPDIdent  if $alpha  is  nonzero;
 $p = $p + 1;
  goto SBPDLoop;
 SBPDStr:
 $p = $p + 1;
 SBPDStrLoop:
  goto SBPDLoop  if $p >=u $end;
 $ch = [$p].bu;
 $t = $ch - 92;
  goto SBPDStrBS  if $t  is  zero;
 $t = $ch - 34;
  goto SBPDStrEnd  if $t  is  zero;
 $p = $p + 1;
  goto SBPDStrLoop;
 SBPDStrBS:
 $p = $p + 2;
  goto SBPDStrLoop;
 SBPDStrEnd:
 $p = $p + 1;
  goto SBPDLoop;
 SBPDHash:
 $t = $p + 1;
  goto SBPDCmt  if $t >=u $end;
 $ch = [$t].bu;
 $digit =  call IsHexDigit($ch);
  goto SBPDHex  if $digit  is  nonzero;
 SBPDCmt:
 $p = $p + 1;
 SBPDCmtLoop:
  goto SBPDLoop  if $p >=u $end;
 $ch = [$p].bu;
```

```
$t = $ch - '\n';
 goto SBPDCmtEnd  if $t  is  zero;
$p = $p + 1;
 goto SBPDCmtLoop;
SBPDCmtEnd:
$p = $p + 1;
 goto SBPDLoop;
SBPDHex:
$p = $p + 1;
SBPDHexLoop:
 goto SBPDLoop  if $p >=u $end;
$ch = [$p].bu;
$digit =  call IsHexDigit($ch);
 goto SBPDLoop  if $digit  is  zero;
$p = $p + 1;
 goto SBPDHexLoop;
SBPDIdent:
$start = $p;
SBPDIdentLoop:
$p = $p + 1;
 goto SBPDIdentEnd  if $p >=u $end;
$ch = [$p].bu;
$alpha =  call IsAlnumUS($ch);
 goto SBPDIdentLoop  if $alpha  is  nonzero;
SBPDIdentEnd:
$nlen = $p - $start;
$t = $nlen - 3;
 goto SBPDLoop  if $t  is  nonzero;
$ch = [$start].bu;
$t = $ch - 'd';
 goto SBPDLoop  if $t  is  nonzero;
$scan = $start + 1;
$ch = [$scan].bu;
$t = $ch - 'e';
 goto SBPDLoop  if $t  is  nonzero;
$scan = $start + 2;
$ch = [$scan].bu;
$t = $ch - 'f';
 goto SBPDLoop  if $t  is  nonzero;
$scan = $p;
SBPDSkipSpace1:
 goto SBPDLoop  if $scan >=u $end;
$ch = [$scan].bu;
$t = $ch  cmpu 32;
 goto SBPDHaveName  if $t  is  pos;
$scan = $scan + 1;
 goto SBPDSkipSpace1;
SBPDHaveName:
$alpha =  call IsAlphaUS($ch);
 goto SBPDLoop  if $alpha  is  zero;
$nptr = $scan;
SBPDNameLoop:
$scan = $scan + 1;
 goto SBPDNameEnd  if $scan >=u $end;
$ch = [$scan].bu;
$alpha =  call IsAlnumUS($ch);
 goto SBPDNameLoop  if $alpha  is  nonzero;
SBPDNameEnd:
$nlen = $scan - $nptr;
SBPDSkipSpace2:
 goto SBPDLoop  if $scan >=u $end;
$ch = [$scan].bu;
```

```
 $t = $ch  cmpu 32;
  goto SBPDCheckParen  if $t  is  pos;
 $scan = $scan + 1;
  goto SBPDSkipSpace2;
 SBPDCheckParen:
 $t = $ch - '(';
  goto SBPDLoop  if $t  is  nonzero;
  call AddProcedure($nptr, $nlen, $sec);
 $p = $scan;
  goto SBPDLoop;
 SBPDDone:
  return;
end
```

ScanBodyForConstantDefs records each `const Name = ...` declaration while skipping strings and line comments.

⟨ ***Unique table procedures*** ⟩ ≡

```
def ScanBodyForConstantDefs($bptr, $blen, $sec):
  register $p, $end, $ch, $t, $alpha, $digit, $start, $nlen, $scan;
 $p = $bptr;
 $end = $bptr + $blen;


 SBCDLoop:
  goto SBCDDone  if $p >=u $end;
 $ch = [$p].bu;
 $t = $ch - 34;
  goto SBCDStr  if $t  is  zero;
 $t = $ch - ' #';
  goto SBCDHash  if $t  is  zero;
 $alpha =  call IsAlphaUS($ch);
  goto SBCDIdent  if $alpha  is  nonzero;
 $p = $p + 1;
  goto SBCDLoop;
 SBCDStr:
 $p = $p + 1;
 SBCDStrLoop:
  goto SBCDLoop  if $p >=u $end;
 $ch = [$p].bu;
 $t = $ch - 92;
  goto SBCDStrBS  if $t  is  zero;
 $t = $ch - 34;
  goto SBCDStrEnd  if $t  is  zero;
 $p = $p + 1;
  goto SBCDStrLoop;
 SBCDStrBS:
 $p = $p + 2;
  goto SBCDStrLoop;
 SBCDStrEnd:
 $p = $p + 1;
  goto SBCDLoop;
 SBCDHash:
 $t = $p + 1;
  goto SBCDCmt  if $t >=u $end;
 $ch = [$t].bu;
 $digit =  call IsHexDigit($ch);
  goto SBCDHex  if $digit  is  nonzero;
 SBCDCmt:
```

```
    $p = $p + 1;
    SBCDCmtLoop:
     goto SBCDLoop  if $p >=u $end;
    $ch = [$p].bu;
    $t = $ch - '\n';
     goto SBCDCmtEnd  if $t  is  zero;
    $p = $p + 1;
     goto SBCDCmtLoop;
    SBCDCmtEnd:
    $p = $p + 1;
     goto SBCDLoop;
    SBCDHex:
    $p = $p + 1;
    SBCDHexLoop:
     goto SBCDLoop  if $p >=u $end;
    $ch = [$p].bu;
    $digit =  call IsHexDigit($ch);
     goto SBCDLoop  if $digit  is  zero;
    $p = $p + 1;
     goto SBCDHexLoop;
    SBCDIdent:
    $start = $p;
    SBCDIdentLoop:
    $p = $p + 1;
     goto SBCDIdentDone  if $p >=u $end;
    $ch = [$p].bu;
    $alpha =  call IsAlnumUS($ch);
     goto SBCDIdentDone  if $alpha  is  zero;
     goto SBCDIdentLoop;
    SBCDIdentDone:
    $nlen = $p - $start;
    $t =  call IsConstKeyword($start, $nlen);
     goto SBCDLoop  if $t  is  zero;
    $scan = $p;
    SBCDSkipGap:
     goto SBCDLoop  if $scan >=u $end;
    $ch = [$scan].bu;
    $alpha =  call IsAlphaUS($ch);
     goto SBCDName  if $alpha  is  nonzero;
    $t = $ch  cmpu 32;
     goto SBCDLoop  if $t  is  pos;
    $scan = $scan + 1;
     goto SBCDSkipGap;
    SBCDName:
    $start = $scan;
    SBCDNameLoop:
    $scan = $scan + 1;
     goto SBCDNameDone  if $scan >=u $end;
    $ch = [$scan].bu;
    $alpha =  call IsAlnumUS($ch);
     goto SBCDNameDone  if $alpha  is  zero;
     goto SBCDNameLoop;
    SBCDNameDone:
    $nlen = $scan - $start;
     call AddUniqueConstant($start, $nlen, $sec, 1);
    $p = $scan;
     goto SBCDLoop;
    SBCDDone:
     return;
   end
```

ScanBodyForConstantUses marks references to already-recorded constants while skipping strings, comments, and the declared name in `const Name = ...`.

⟨ ***Unique table procedures*** ⟩ ≡

```
def ScanBodyForConstantUses($bptr, $blen, $sec):
  register $p, $end, $ch, $t, $alpha, $digit, $start, $nlen, $const_kw;
 $p = $bptr;
 $end = $bptr + $blen;


 $const_kw = 0;
 SBCULoop:
  goto SBCUDone  if $p >=u $end;
 $ch = [$p].bu;
 $t = $ch - '\n';
  goto SBCUNewline  if $t  is  zero;
 $t = $ch - ';';
  goto SBCUSemi  if $t  is  zero;
 $t = $ch - 34;
  goto SBCUStr  if $t  is  zero;
 $t = $ch - ' #';
  goto SBCUHash  if $t  is  zero;
 $alpha =  call IsAlphaUS($ch);
  goto SBCUIdent  if $alpha  is  nonzero;
 $p = $p + 1;
  goto SBCULoop;
 SBCUNewline:
 $const_kw = 0;
 $p = $p + 1;
  goto SBCULoop;
 SBCUSemi:
 $const_kw = 0;
 $p = $p + 1;
  goto SBCULoop;
 SBCUStr:
 $p = $p + 1;
 SBCUStrLoop:
  goto SBCULoop  if $p >=u $end;
 $ch = [$p].bu;
 $t = $ch - 92;
  goto SBCUStrBS  if $t  is  zero;
 $t = $ch - 34;
  goto SBCUStrEnd  if $t  is  zero;
 $p = $p + 1;
  goto SBCUStrLoop;
 SBCUStrBS:
 $p = $p + 2;
  goto SBCUStrLoop;
 SBCUStrEnd:
 $p = $p + 1;
  goto SBCULoop;
 SBCUHash:
 $t = $p + 1;
  goto SBCUCmt  if $t >=u $end;
 $ch = [$t].bu;
 $digit =  call IsHexDigit($ch);
  goto SBCUHex  if $digit  is  nonzero;
 SBCUCmt:
 $p = $p + 1;
 SBCUCmtLoop:
```

```
    goto SBCULoop  if $p >=u $end;
  $ch = [$p].bu;
  $t = $ch - '\n';
    goto SBCUCmtEnd  if $t  is  zero;
  $p = $p + 1;
    goto SBCUCmtLoop;
  SBCUCmtEnd:
  $p = $p + 1;
    goto SBCULoop;
  SBCUHex:
  $p = $p + 1;
  SBCUHexLoop:
    goto SBCULoop  if $p >=u $end;
  $ch = [$p].bu;
  $digit =  call IsHexDigit($ch);
    goto SBCULoop  if $digit  is  zero;
  $p = $p + 1;
    goto SBCUHexLoop;
  SBCUIdent:
  $start = $p;
  SBCUIdentLoop:
  $p = $p + 1;
    goto SBCUIdentDone  if $p >=u $end;
  $ch = [$p].bu;
  $alpha =  call IsAlnumUS($ch);
    goto SBCUIdentDone  if $alpha  is  zero;
    goto SBCUIdentLoop;
  SBCUIdentDone:
  $nlen = $p - $start;
  $t = $const_kw;
    goto SBCUCheckKeyword  if $t  is  zero;
  $const_kw = 0;
    goto SBCULoop;
  SBCUCheckKeyword:
  $t =  call IsConstKeyword($start, $nlen);
    goto SBCUCheckKnown  if $t  is  zero;
  $const_kw = 1;
    goto SBCULoop;
  SBCUCheckKnown:
  $t =  call FindConstantByName($start, $nlen);
    goto SBCULoop  if $t  is  neg;
    call AddUniqueConstant($start, $nlen, $sec, 0);
    goto SBCULoop;
  SBCUDone:
    return;
  end
```

BuildUniqueTables is the driver. It walks the parsed section table, maintains the running current section number, and dispatches each code chunk to the updaters above. Chunk-reference sigils inside code bodies are also scanned here to record uses in UniqueChunks.

⟨ ***Unique table procedures*** ⟩ ≡

```
def BuildUniqueTables():
  register $i, $entry, $kind, $hptr, $hlen, $tptr, $tlen, $num, $cur, $ord;
  register $p, $end, $ch, $t, $ns, $nr, $refp, $refl;
 $cur = 0;


 $i = 0;
 BULoop:
  goto BUDone  if $i >=u $sec_cnt;
 $entry =  call SectionEntryAt($i);
 $kind = [$entry].o;
 $num  = [$entry + SEC_NUM].o;
  goto BUCode  if $kind  is  nonzero;
  goto BUNext  if $num  is  zero;
 $cur = $num;
 $total_sections = $cur;
  goto BUNext;
 BUCode:
 $hptr = [$entry + SEC_HPTR].o;
 $hlen = [$entry + SEC_HLEN].o;
 $tptr = [$entry + SEC_TPTR].o;
 $tlen = [$entry + SEC_TLEN].o;
 $ord =  call AddUniqueChunk($hptr, $hlen, $cur, 1);
 [$entry + SEC_NUM].o = $ord;
 $p = $tptr;
 $end = $tptr + $tlen;
 BURefLoop:
  goto BURefsDone  if $p >=u $end;
 $ch = [$p].bu;
 $t = $ch - '@';
  goto BURefAdv  if $t  is  nonzero;
 $t = $p + 1;
  goto BURefAdv  if $t >=u $end;
 $ch = [$t].bu;
 $t = $ch - '<';
  goto BURefAdv  if $t  is  nonzero;
 $ns = $p + 2;
 $nr = 0;
 BURefName:
 $t = $ns + $nr;
  goto BURefsDone  if $t >=u $end;
 $ch = [$t].bu;
 $t = $ch - '@';
  goto BURefNameInc  if $t  is  nonzero;
 $t = $ns + $nr;
 $t = $t + 1;
  goto BURefsDone  if $t >=u $end;
 $ch = [$t].bu;
 $t = $ch - '>';
  goto BURefNameInc  if $t  is  nonzero;
 $refp = $RefNameBuf;
 $refl =  call NormNameTo($ns, $nr, $refp);
  call AddUniqueChunk($refp, $refl, $cur, 0);
 $p = $ns + $nr;
 $p = $p + 2;
  goto BURefLoop;
 BURefNameInc:
 $nr = $nr + 1;
  goto BURefName;
 BURefAdv:
 $p = $p + 1;
```

```
    goto BURefLoop;
  BURefsDone:
    call ScanBodyForProcedureDefs($tptr, $tlen, $cur);
    call ScanBodyForLabelsRegs($tptr, $tlen, $cur);
    call ScanBodyForConstantDefs($tptr, $tlen, $cur);
  BUNext:
  $i = $i + 1;
    goto BULoop;
  BUDone:
  $cur = 0;
  $i = 0;
  BUConstLoop:
    goto BUConstDone  if $i >=u $sec_cnt;
  $entry =  call SectionEntryAt($i);
  $kind = [$entry].o;
  $num  = [$entry + SEC_NUM].o;
    goto BUConstCode  if $kind  is  nonzero;
    goto BUConstNext  if $num  is  zero;
  $cur = $num;
    goto BUConstNext;
  BUConstCode:
  $tptr = [$entry + SEC_TPTR].o;
  $tlen = [$entry + SEC_TLEN].o;
    call ScanBodyForConstantUses($tptr, $tlen, $cur);
  BUConstNext:
  $i = $i + 1;
    goto BUConstLoop;
  BUConstDone:
    return;
end
```

13. Sorting cross-reference tables.

Chunk names, labels, and constants are reported alphabetically in the cross-reference index. Registers are reported numerically. All four sorts use a shift-based insertion sort: the pivot entry is copied to `SortKeyBuf`, larger predecessors are shifted one slot to the right (a single octa-by-octa block move per shift, not a swap), and the pivot is finally written into the gap. This avoids the doubled data movement of a swap-based insertion sort while remaining tiny in code size, and it also reuses one helper across all entry sizes.

CopyEntry copies *esize* bytes (always a multiple of 8) one octa at a time from *src* to *dst*. It is the only data-movement primitive used by the sorts.

⟨ ***Sorting procedures*** ⟩ ≡

```
def CopyEntry($dst, $src, $esize):
  register $k, $t, $off;
 $k = 0;
 CELoop:
 $t = $k  cmpu $esize;
  goto CEDone  if $t  is  nonneg;
 $off = $src + $k;
 $t = [$off].o;
 $off = $dst + $k;
 [$off].o = $t;
 $k = $k + 8;
  goto CELoop;
 CEDone:
```

```
  return;
end
```

SortOneTable is a shift-based insertion sort over the StrCmp key formed by the (name-ptr, name-len) pair at the start of each entry. The chunk, label, constant, and procedure tables share this layout.

⟨ ***Sorting procedures*** ⟩ ≡

```
def SortOneTable($table, $count, $esize):
  register $i, $j, $pj, $pp, $key, $ap, $al, $bp, $bl, $cmp, $tmp;
 $key = $SortKeyBuf;
 $i = 1;


 SOLoop:
 $cmp = $i  cmpu $count;
  goto SODone  if $cmp  is  nonneg;
  # Save the pivot entry table[i] in $key.
 $pj = $i * $esize;
 $pj = $pj + $table;
  call CopyEntry($key, $pj, $esize);
 $bp = [$key].o;
 $bl = [$key + 8].o;
  # Shift greater predecessors one slot to the right.
 $j = $i;
 SOInner:
  goto SOPlace  if $j  is  zero;
 $tmp = $j - 1;
 $pp = $tmp * $esize;
 $pp = $pp + $table;
 $ap = [$pp].o;
 $al = [$pp + 8].o;
 $cmp =  call StrCmp($ap, $al, $bp, $bl);
  goto SOPlace  if $cmp  is  nonpos;
 $pj = $j * $esize;
 $pj = $pj + $table;
  call CopyEntry($pj, $pp, $esize);
 $j = $j - 1;
  goto SOInner;
 SOPlace:
 $pj = $j * $esize;
 $pj = $pj + $table;
  call CopyEntry($pj, $key, $esize);
 $i = $i + 1;
  goto SOLoop;
 SODone:
  return;
end
```

⟨ ***Sorting procedures*** ⟩ ≡

```
def SortUniqueChunks():
  call SortOneTable($UniqueChunks, $uc_cnt, UNIQ_ENTRY_SIZE);
  return;
end
```

⟨ ***Sorting procedures*** ⟩ ≡

```
def SortLabels():
  call SortOneTable($LabelTable, $lbl_cnt, LBL_ENTRY_SIZE);
  return;
end
```

⟨ ***Sorting procedures*** ⟩ ≡

```
def SortConstants():
  call SortOneTable($ConstantTable, $const_cnt, CONST_ENTRY_SIZE);
  # The sort reorders entries beneath |ConstHashTable|, so the
  # stored indices are now stale.  Rebuild before any further
  # |FindConstantByName| call.
  call HashRebuild($ConstHashTable, CONST_HASH_SIZE, CONST_HASH_MASK,
                   $ConstantTable, $const_cnt, CONST_ENTRY_SIZE);
  return;
end
```

⟨ ***Sorting procedures*** ⟩ ≡

```
def SortProcedures():
  call SortOneTable($ProcedureTable, $proc_cnt, PROC_ENTRY_SIZE);
  return;
end
```

14. Output helpers.

OutputBuf accumulates the entire LaTeX document. AppendByte is the hot path; AppendBytes, AppendCStr, AppendNewline, and AppendInt are convenience wrappers. AppendPLWRaw copies a byte slice while mapping every at-at escape back to a single at-sign. AppendPLWTeXEscaped applies that same doubled-at decoding before escaping the TeX specials needed by prose inline code, and AppendVerbatimEscaped emits a slice with only the three fancyvrb commandchars escaped.

⟨ ***Output helpers*** ⟩ ≡

```
def AppendByte($ch):
  register $tmp;
 $tmp = $out_end  cmpu OUT_MAX;
  if $tmp  is  nonneg:
    call ErrHalt("plmmix-weave: output buffer full\n");
  end
 $OutputBuf[$out_end].bu = $ch;
 $out_end = $out_end + 1;
  return;
end
```

⟨ ***Output helpers*** ⟩ ≡

```
def AppendBytes($src, $len):
  register $ch, $dst, $end_after, $tmp;
  goto ABDone  if $len  is  zero;
  # Bound-check once for the whole span, then copy without
  # per-byte calls to AppendByte.  This is the dominant write path
  # (every AppendCStr ends up here too), so the saved call/return
  # plus per-byte cmpu pays off.
 $end_after = $out_end + $len;
 $tmp = $end_after  cmpu OUT_MAX;
  if $tmp  is  pos:
    call ErrHalt("plmmix-weave: output buffer full\n");
  end
 $dst = $OutputBuf + $out_end;
 ABLoop:
  goto ABDoneStore  if $len  is  zero;
 $ch = [$src].bu;
 [$dst].bu = $ch;
 $src = $src + 1;
 $dst = $dst + 1;
 $len = $len - 1;
  goto ABLoop;
 ABDoneStore:
 $out_end = $end_after;
 ABDone:
  return;
end
```

⟨ ***Output helpers*** ⟩ ≡

```
def AppendCStr($ptr):
  register $ch, $tmp, $dst, $limit;
  # Hoist |OUT_MAX| and the destination pointer out of the loop.  The
  # previous form recomputed the 22-bit |OUT_MAX| constant (four
  # |SET*| instructions) on every byte; this is the single hottest
  # write path so per-byte overhead matters.
 $dst   = $OutputBuf + $out_end;
 $limit = $OutputBuf + OUT_MAX;
 ACSLoop:
 $ch = [$ptr].bu;
  if $ch  is  zero:
   $out_end = $dst - $OutputBuf;
    return;
  end
 $tmp = $dst  cmpu $limit;
  if $tmp  is  nonneg:
    call ErrHalt("plmmix-weave: output buffer full\n");
  end
 [$dst].bu = $ch;
 $dst = $dst + 1;
 $ptr = $ptr + 1;
  goto ACSLoop;
end
```

⟨ ***Output helpers*** ⟩ ≡

```
def AppendNewline():
  register $t;
 $t = $out_end  cmpu OUT_MAX;
  if $t  is  nonneg:
    call ErrHalt("plmmix-weave: output buffer full\n");
  end
 $OutputBuf[$out_end].bu = '\n';
 $out_end = $out_end + 1;
  return;
end
```

⟨ ***Output helpers*** ⟩ ≡

```
def AppendInt($n):
  register $count, $digit, $tmp, $ptr;
  goto AIWork  if $n  is  nonzero;
  call AppendByte('0');


  return;
 AIWork:
 $count = 0;
 AILoop:
  goto AIEmit  if $n  is  zero;
 $tmp = $n / 10;
 $digit = $tmp * 10;
 $digit = $n - $digit;
 $digit = $digit + '0';
 $ptr = $IntBuf + $count;
 [$ptr].bu = $digit;
 $count = $count + 1;
 $n = $tmp;
  goto AILoop;
 AIEmit:
  goto AIDone  if $count  is  zero;
 $count = $count - 1;
 $ptr = $IntBuf + $count;
 $digit = [$ptr].bu;
  call AppendByte($digit);
  goto AIEmit;
 AIDone:
  return;
end
```

AppendPLWRaw emits a raw slice, converting each at-at escape back to a single at-sign. Used to emit the undecorated body content inside PLMCode blocks; kept as a helper for future tools that may want a raw pass-through of body text.

⟨ ***Output helpers*** ⟩ ≡

```
def AppendPLWRaw($ptr, $len):
  register $p, $end, $ch, $t;
 $p = $ptr;
 $end = $ptr + $len;


 APRLoop:
  goto APRDone  if $p >=u $end;
 $ch = [$p].bu;
 $t = $ch - '@';
  goto APREmit  if $t  is  nonzero;
 $t = $p + 1;
  goto APREmit  if $t >=u $end;
 $ch = [$t].bu;
 $t = $ch - '@';
  goto APREmitAt  if $t  is  zero;
 $ch = '@';
  call AppendByte($ch);
 $p = $p + 1;
  goto APRLoop;
 APREmitAt:
  call AppendByte('@');
 $p = $p + 2;
  goto APRLoop;
 APREmit:
  call AppendByte($ch);
 $p = $p + 1;
  goto APRLoop;
 APRDone:
  return;
end
```

AppendPLWTeXEscaped first decodes each doubled at-sign to a single literal at-sign, then applies the usual TeX escaping. It is used for prose fragments such as `\.{...}` and `\.{\.\{...\}}` that need both transformations.

⟨ ***Output helpers*** ⟩ ≡

```
def AppendPLWTeXEscaped($ptr, $len):
  register $p, $end, $ch, $t;
 $p = $ptr;
 $end = $ptr + $len;


 APTELoop:
  goto APTEDone  if $p >=u $end;
 $ch = [$p].bu;
 $t = $ch - '@';
  goto APTEChar  if $t  is  nonzero;
 $t = $p + 1;
  goto APTEEmitAt  if $t >=u $end;
 $ch = [$t].bu;
 $t = $ch - '@';
  goto APTEEmitAt  if $t  is  nonzero;
 $ch = '@';
 $p = $p + 2;
  goto APTEEmit;
 APTEEmitAt:
 $ch = '@';
```

```
$p = $p + 1;
 goto APTEEmit;
APTEChar:
$p = $p + 1;
APTEEmit:
$t = $ch - 92;
 goto APTEBS  if $t  is  zero;
$t = $ch - '{';
 goto APTELB  if $t  is  zero;
$t = $ch - '}';
 goto APTERB  if $t  is  zero;
$t = $ch - '$';
 goto APTEDol  if $t  is  zero;
$t = $ch - '&';
 goto APTEAmp  if $t  is  zero;
$t = $ch - '%';
 goto APTEPct  if $t  is  zero;
$t = $ch - ' #';
 goto APTEHash  if $t  is  zero;
$t = $ch - '_';
 goto APTEUnd  if $t  is  zero;
$t = $ch - '^';
 goto APTECar  if $t  is  zero;
$t = $ch - '~';
 goto APTETil  if $t  is  zero;
 # Inline AppendByte: bound-check then store.
$t = $out_end  cmpu OUT_MAX;
 if $t  is  nonneg:
   call ErrHalt("plmmix-weave: output buffer full\n");
 end
$OutputBuf[$out_end].bu = $ch;
$out_end = $out_end + 1;
 goto APTELoop;
APTEBS:
 call AppendCStr(&StrBackslashTb);
 goto APTELoop;
APTELB:
 call AppendCStr(&StrBraceLOpen);
 goto APTELoop;
APTERB:
 call AppendCStr(&StrBraceLClose);
 goto APTELoop;
APTEDol:
 call AppendCStr(&StrDollar);
 goto APTELoop;
APTEAmp:
 call AppendCStr(&StrAmp);
 goto APTELoop;
APTEPct:
 call AppendCStr(&StrPercent);
 goto APTELoop;
APTEHash:
 call AppendCStr(&StrHash);
 goto APTELoop;
APTEUnd:
 call AppendCStr(&StrUnder);
 goto APTELoop;
APTECar:
 call AppendCStr(&StrCaret);
 goto APTELoop;
APTETil:
 call AppendCStr(&StrTilde);
```

```
    goto APTELoop;
  APTEDone:
    return;
  end
```

AppendTeXEscaped escapes the nine TeX specials listed in the weave preamble and verbatim helpers.

⟨ ***Output helpers*** ⟩ ≡

```
def AppendTeXEscaped($ptr, $len):
  register $p, $end, $ch, $t;
 $p = $ptr;
 $end = $ptr + $len;


 ATELoop:
  goto ATEDone  if $p >=u $end;
 $ch = [$p].bu;
 $p = $p + 1;
 $t = $ch - 92;
  goto ATEBS  if $t  is  zero;
 $t = $ch - '{';
  goto ATELB  if $t  is  zero;
 $t = $ch - '}';
  goto ATERB  if $t  is  zero;
 $t = $ch - '$';
  goto ATEDol  if $t  is  zero;
 $t = $ch - '&';
  goto ATEAmp  if $t  is  zero;
 $t = $ch - '%';
  goto ATEPct  if $t  is  zero;
 $t = $ch - ' #';
  goto ATEHash  if $t  is  zero;
 $t = $ch - '_';
  goto ATEUnd  if $t  is  zero;
 $t = $ch - '^';
  goto ATECar  if $t  is  zero;
 $t = $ch - '~';
  goto ATETil  if $t  is  zero;
  # Inline AppendByte: bound-check then store.
 $t = $out_end  cmpu OUT_MAX;
  if $t  is  nonneg:
    call ErrHalt("plmmix-weave: output buffer full\n");
  end
 $OutputBuf[$out_end].bu = $ch;
 $out_end = $out_end + 1;
  goto ATELoop;
 ATEBS:
  call AppendCStr(&StrBackslashTb);
  goto ATELoop;
 ATELB:
  call AppendCStr(&StrBraceLOpen);
  goto ATELoop;
 ATERB:
  call AppendCStr(&StrBraceLClose);
  goto ATELoop;
 ATEDol:
  call AppendCStr(&StrDollar);
  goto ATELoop;
```

```
 ATEAmp:
  call AppendCStr(&StrAmp);
  goto ATELoop;
 ATEPct:
  call AppendCStr(&StrPercent);
  goto ATELoop;
 ATEHash:
  call AppendCStr(&StrHash);
  goto ATELoop;
 ATEUnd:
  call AppendCStr(&StrUnder);
  goto ATELoop;
 ATECar:
  call AppendCStr(&StrCaret);
  goto ATELoop;
 ATETil:
  call AppendCStr(&StrTilde);
  goto ATELoop;
 ATEDone:
  return;
end
```

AppendVerbatimEscaped emits a byte slice with only the three fancyvrb commandchars escaped. All other bytes are printed literally.

⟨ ***Output helpers*** ⟩ ≡

```
def AppendVerbatimEscaped($ptr, $len):
  register $p, $end, $ch, $t;
 $p = $ptr;
 $end = $ptr + $len;


 AVELoop:
  goto AVEDone  if $p >=u $end;
 $ch = [$p].bu;
 $p = $p + 1;
 $t = $ch - 92;
  goto AVEBS  if $t  is  zero;
 $t = $ch - '{';
  goto AVELB  if $t  is  zero;
 $t = $ch - '}';
  goto AVERB  if $t  is  zero;
  # Inline AppendByte: bound-check then store.
 $t = $out_end  cmpu OUT_MAX;
  if $t  is  nonneg:
    call ErrHalt("plmmix-weave: output buffer full\n");
  end
 $OutputBuf[$out_end].bu = $ch;
 $out_end = $out_end + 1;
  goto AVELoop;
 AVEBS:
  call AppendCStr(&StrBackslashTb);
  goto AVELoop;
 AVELB:
  call AppendCStr(&StrBraceLOpen);
  goto AVELoop;
 AVERB:
  call AppendCStr(&StrBraceLClose);
  goto AVELoop;
```

```
 AVEDone:
  return;
end
```

15. Anchor emitters.

Anchors are strings used as `\hypertarget` keys and `\hyperlink` targets. Two shapes are used: `plmchunk.<safe-name>` for chunk anchors and `plmsec.<n>` for section anchors. The chunk-anchor builder emits a sanitized form of the name in which every non-alphanumeric byte is replaced by a dot.

For example, the root chunk `Program` produces the anchor `plmchunk.Program`, the chunk `ParsePLW: main scan loop` produces `plmchunk.ParsePLW...main.scan.loop`, and the major section numbered 4 produces `plmsec.4`.

⟨ ***Anchor emitters*** ⟩ ≡

```
def EmitSafeAnchorText($ptr, $len):
  register $p, $end, $ch, $a, $d;
 $p = $ptr;
 $end = $ptr + $len;


 ESATLoop:
  goto ESATDone  if $p >=u $end;
 $ch = [$p].bu;
 $p = $p + 1;
 $d =  call IsDigit($ch);
  goto ESATEmit  if $d  is  nonzero;
 $a = $ch - 'A';
  goto ESATCheckLo  if $a  is  neg;
 $a = $ch - 'Z';
  goto ESATEmit  if $a  is  nonpos;
 ESATCheckLo:
 $a = $ch - 'a';
  goto ESATDot  if $a  is  neg;
 $a = $ch - 'z';
  goto ESATEmit  if $a  is  nonpos;
 ESATDot:
  call AppendByte('.');
  goto ESATLoop;
 ESATEmit:
  call AppendByte($ch);
  goto ESATLoop;
 ESATDone:
  return;
end
```

⟨ ***Anchor emitters*** ⟩ ≡

```
def EmitChunkAnchor($nptr, $nlen):
  call AppendCStr(&StrChunkAnchor);
  call EmitSafeAnchorText($nptr, $nlen);
  return;
end
```

⟨ ***Anchor emitters*** ⟩ ≡

```
def EmitLabelAnchor($nptr, $nlen):
  call AppendCStr(&StrLabelAnchor);
  call EmitSafeAnchorText($nptr, $nlen);
  return;
end
```

⟨ ***Anchor emitters*** ⟩ ≡

```
def EmitConstAnchor($nptr, $nlen):
  call AppendCStr(&StrConstAnchor);
  call EmitSafeAnchorText($nptr, $nlen);
  return;
end
```

⟨ ***Anchor emitters*** ⟩ ≡

```
def EmitProcAnchor($nptr, $nlen):
  call AppendCStr(&StrProcAnchor);
  call EmitSafeAnchorText($nptr, $nlen);
  return;
end
```

⟨ ***Anchor emitters*** ⟩ ≡

```
def EmitSecAnchor($num):
  call AppendCStr(&StrSecAnchor);
  call AppendInt($num);
  return;
end
```

EmitSecLink emits a section link of the form `\hyperlink{plmsec.N}{\S N}`. If the argument is zero it emits the literal text `(preamble)` instead, denoting the unnumbered preamble section.

⟨ ***Anchor emitters*** ⟩ ≡

```
def EmitSecLink($num):
  goto ESLPre  if $num  is  zero;
  call AppendCStr(&StrHyperlink);
  call EmitSecAnchor($num);
  call AppendCStr(&StrBraceClose);
  call AppendCStr(&StrBraceOpen);
  call AppendCStr(&StrSS);
  call AppendInt($num);
  call AppendCStr(&StrBraceClose);
  return;
 ESLPre:
  call AppendCStr(&StrPreamble);
  return;
end
```

16. Rendering prose.

Prose text in a PLW file is treated as raw LaTeX source. Four kinds of inline markup are converted: `...` becomes exttt\texttt{...} with TeX-escaped content, the CWEB sequence `\.{...}` becomes the same, and chunk references of the form `@<name@>` become hyperlinking exttt\plmref calls, and section references of the form `\S 9` become hyperlinks to the corresponding section anchor. Within ordinary prose, `@@` is decoded to a single literal at-sign before output. Verbatim environments are emitted untouched, so the input source can contain verbatim content verbatim.

⟨ ***Prose rendering*** ⟩ ≡

```
def IsProcedureName($nptr, $nlen) -> $r:
  register $idx;
 $r = 0;
 $idx =  call FindProcedureByName($nptr, $nlen);
  goto IPNDone  if $idx  is  neg;
 $r = 1;
 IPNDone:
  return $r;
end
```

⟨ ***Prose rendering*** ⟩ ≡

```
def EmitProcedureTextRef($nptr, $nlen):
  call AppendCStr(&StrHyperlink);
  call EmitProcAnchor($nptr, $nlen);
  call AppendCStr(&StrBraceClose);
  call AppendCStr(&StrBraceOpen);
  call AppendCStr(&StrTexttt);
  call AppendPLWTeXEscaped($nptr, $nlen);
  call AppendCStr(&StrBraceClose);
  call AppendCStr(&StrBraceClose);
  return;
end
```

⟨ ***Prose rendering*** ⟩ ≡

```
def IsConstantName($nptr, $nlen) -> $r:
  register $idx;
 $r = 0;
 $idx =  call FindConstantByName($nptr, $nlen);
  goto ICNDone  if $idx  is  neg;
 $r = 1;
 ICNDone:
  return $r;
end
```

⟨ ***Prose rendering*** ⟩ ≡

```
def EmitConstantTextRef($nptr, $nlen):
  call AppendCStr(&StrHyperlink);
  call EmitConstAnchor($nptr, $nlen);
  call AppendCStr(&StrBraceClose);
  call AppendCStr(&StrBraceOpen);
  call AppendCStr(&StrTexttt);
  call AppendPLWTeXEscaped($nptr, $nlen);
  call AppendCStr(&StrBraceClose);
  call AppendCStr(&StrBraceClose);
  return;
end
```

⟨ ***Prose rendering*** ⟩ ≡

```
def IsLabelName($nptr, $nlen) -> $r:
  register $idx;
 $r = 0;
 $idx =  call FindLabelByName($nptr, $nlen);
  goto ILNDone  if $idx  is  neg;
 $r = 1;
 ILNDone:
  return $r;
end
```

⟨ ***Prose rendering*** ⟩ ≡

```
def RenderProseInline($ptr, $len):
  register $p, $end, $ch, $t, $tok_start, $tok_end, $inner_ptr, $inner_len, $delim, $nxt, $ns,
    $nr, $refp, $refl, $sec, $scan, $saw;
 $p = $ptr;
 $end = $ptr + $len;


 RPILoop:
  goto RPIDone  if $p >=u $end;
 $ch = [$p].bu;
 $t = $ch - '@';
  goto RPIAt  if $t  is  zero;
 $t = $ch - '|';
  goto RPIBar  if $t  is  zero;
 $t = $ch - 92;
  goto RPIBackslash  if $t  is  zero;
  call AppendByte($ch);
```

```
$p = $p + 1;
 goto RPILoop;
RPIAt:
$t = $p + 1;
 goto RPIAtLit  if $t >=u $end;
$ch = [$t].bu;
$nxt = $ch;
$t = $ch - '@';
 goto RPIAtAt  if $t  is  zero;
$t = $nxt - '<';
 goto RPIAtAng  if $t  is  zero;
 goto RPIAtLit;
RPIAtAt:
 call AppendByte('@');
$p = $p + 2;
 goto RPILoop;
RPIAtAng:
$ns = $p + 2;
$nr = 0;
RPIAtScan:
$t = $ns + $nr;
 goto RPIAtLit  if $t >=u $end;
$ch = [$t].bu;
$t = $ch - '@';
 goto RPIAtInc  if $t  is  nonzero;
$t = $ns + $nr;
$t = $t + 1;
 goto RPIAtLit  if $t >=u $end;
$ch = [$t].bu;
$t = $ch - '>';
 goto RPIAtInc  if $t  is  nonzero;
$refp = $RefNameBuf;
$refl =  call NormNameTo($ns, $nr, $refp);
 call EmitChunkRef($refp, $refl);
$p = $ns + $nr;
$p = $p + 2;
 goto RPILoop;
RPIAtInc:
$nr = $nr + 1;
 goto RPIAtScan;
RPIAtLit:
 call AppendByte('@');
$p = $p + 1;
 goto RPILoop;
RPIBar:
$tok_start = $p + 1;
$tok_end = $tok_start;
RPIBarScan:
 goto RPIBarNone  if $tok_end >=u $end;
$ch = [$tok_end].bu;
$t = $ch - '\n';
 goto RPIBarNone  if $t  is  zero;
$t = $ch - '|';
 goto RPIBarFound  if $t  is  zero;
$tok_end = $tok_end + 1;
 goto RPIBarScan;
RPIBarNone:
 call AppendByte('|');
$p = $p + 1;
 goto RPILoop;
RPIBarFound:
$inner_ptr = $tok_start;
```

```
$inner_len = $tok_end - $tok_start;
$t =  call IsProcedureName($inner_ptr, $inner_len);
 goto RPIBarCheckConst  if $t  is  zero;
 call EmitProcedureTextRef($inner_ptr, $inner_len);
$p = $tok_end + 1;
 goto RPILoop;
RPIBarCheckConst:
$t =  call IsConstantName($inner_ptr, $inner_len);
 goto RPIBarPlain  if $t  is  zero;
 call EmitConstantTextRef($inner_ptr, $inner_len);
$p = $tok_end + 1;
 goto RPILoop;
RPIBarPlain:
 call AppendCStr(&StrTexttt);
 call AppendPLWTeXEscaped($inner_ptr, $inner_len);
 call AppendCStr(&StrBraceClose);
$p = $tok_end + 1;
 goto RPILoop;
RPIBackslash:
$t = $p + 1;
 goto RPINotSec  if $t >=u $end;
$ch = [$t].bu;
$t = $ch - 'S';
 goto RPINotSec  if $t  is  nonzero;
$scan = $p + 2;
RPISecSpaces:
 goto RPISecDigits  if $scan >=u $end;
$ch = [$scan].bu;
$t = $ch - ' ';
 goto RPISecDigits  if $t  is  nonzero;
$scan = $scan + 1;
 goto RPISecSpaces;
RPISecDigits:
$sec = 0;
$saw = 0;
RPISecLoop:
 goto RPISecEmit  if $scan >=u $end;
$ch = [$scan].bu;
$t =  call IsDigit($ch);
 goto RPISecEmit  if $t  is  zero;
$sec = $sec * 10;
$t = $ch - '0';
$sec = $sec + $t;
$saw = 1;
$scan = $scan + 1;
 goto RPISecLoop;
RPISecEmit:
 goto RPINotSec  if $saw  is  zero;
 call EmitSecLink($sec);
$p = $scan;
 goto RPILoop;
RPINotSec:
$tok_start = $p;
$t = $p + 5;
 goto RPINotVerb  if $t >=u $end;
$ch = [$p + 1].bu;
$t = $ch - 'v';
 goto RPINotVerb  if $t  is  nonzero;
$ch = [$p + 2].bu;
$t = $ch - 'e';
 goto RPINotVerb  if $t  is  nonzero;
$ch = [$p + 3].bu;
```

```
$t = $ch - 'r';
 goto RPINotVerb  if $t  is  nonzero;
$ch = [$p + 4].bu;
$t = $ch - 'b';
 goto RPINotVerb  if $t  is  nonzero;
$tok_end = $p + 5;
 goto RPIVerbDelim  if $tok_end >=u $end;
$ch = [$tok_end].bu;
$t = $ch - '*';
 goto RPIVerbDelim  if $t  is  nonzero;
$tok_end = $tok_end + 1;
RPIVerbDelim:
 goto RPINotVerb  if $tok_end >=u $end;
$delim = [$tok_end].bu;
$t = $delim - 10;
 goto RPINotVerb  if $t  is  zero;
$tok_end = $tok_end + 1;
RPIVerbScan:
 goto RPINotVerb  if $tok_end >=u $end;
$ch = [$tok_end].bu;
$t = $ch - 10;
 goto RPINotVerb  if $t  is  zero;
$t = $ch - $delim;
 goto RPIVerbFound  if $t  is  zero;
$tok_end = $tok_end + 1;
 goto RPIVerbScan;
RPIVerbFound:
$tok_end = $tok_end + 1;
$inner_len = $tok_end - $tok_start;
 call AppendBytes($tok_start, $inner_len);
$p = $tok_end;
 goto RPILoop;
RPINotVerb:
$t = $p + 1;
 goto RPIPlainBS  if $t >=u $end;
$ch = [$t].bu;
$t = $ch - '.';
 goto RPIPlainBS  if $t  is  nonzero;
$t = $p + 2;
 goto RPIPlainBS  if $t >=u $end;
$ch = [$t].bu;
$t = $ch - '{';
 goto RPIPlainBS  if $t  is  nonzero;
$tok_start = $p + 3;
$tok_end = $tok_start;
RPICScan:
 goto RPIPlainBS  if $tok_end >=u $end;
$ch = [$tok_end].bu;
$t = $ch - '}';
 goto RPICFound  if $t  is  zero;
$t = $ch - '{';
 goto RPIPlainBS  if $t  is  zero;
$tok_end = $tok_end + 1;
 goto RPICScan;
RPICFound:
$inner_ptr = $tok_start;
$inner_len = $tok_end - $tok_start;
$t =  call IsProcedureName($inner_ptr, $inner_len);
 goto RPICCheckConst  if $t  is  zero;
 call EmitProcedureTextRef($inner_ptr, $inner_len);
$p = $tok_end + 1;
 goto RPILoop;
```

```
 RPICCheckConst:
 $t =  call IsConstantName($inner_ptr, $inner_len);
  goto RPICPlain  if $t  is  zero;
  call EmitConstantTextRef($inner_ptr, $inner_len);
 $p = $tok_end + 1;
  goto RPILoop;
 RPICPlain:
  call AppendCStr(&StrTexttt);
  call AppendPLWTeXEscaped($inner_ptr, $inner_len);
  call AppendCStr(&StrBraceClose);
 $p = $tok_end + 1;
  goto RPILoop;
 RPIPlainBS:
  call AppendByte(92);
 $p = $p + 1;
  goto RPILoop;
 RPIDone:
  return;
end
```

RenderProse walks a prose slice line by line, toggling a verbatim state flag whenever a begin-verbatim or end-verbatim marker appears on a line. Lines inside a verbatim block are emitted unchanged; other lines go through RenderProseInline.

⟨ ***Prose rendering*** ⟩ ≡

```
def RenderProse($ptr, $len):
  register $p, $end, $line_start, $line_end, $ch, $t, $in_verb, $has_end, $has_beg;
 $p = $ptr;
 $end = $ptr + $len;


 $in_verb = 0;
 RPLoop:
  goto RPDone  if $p >=u $end;
 $line_start = $p;
 RPScan:
  goto RPEOL  if $p >=u $end;
 $ch = [$p].bu;
 $t = $ch - '\n';
  goto RPNL  if $t  is  zero;
 $p = $p + 1;
  goto RPScan;
 RPNL:
 $p = $p + 1;
 RPEOL:
 $line_end = $p;
 $t = $line_end - $line_start;
  # Only one of the two needles is meaningful per state, so we skip
  # the irrelevant ContainsSliceCStr scan on each line.  When inside
  # a verbatim block, lines without the end-marker render raw
  # without scanning for begin (a nested begin would not change
  # state anyway).
  goto RPInVerb  if $in_verb  is  nonzero;
  # State: outside verbatim.
 $has_beg =  call MatchIncludeLine($line_start, $t);
  goto RPNotInclude  if $has_beg  is  zero;
  call AppendCStr(&StrBeginPLM);
  call AppendVerbatimEscaped($line_start, $t);
  goto RPIncludeHasNL  if $t  is  zero;
```

```
    $ch = [$line_end - 1].bu;
    $has_end = $ch - '\n';
     goto RPIncludeHasNL  if $has_end  is  zero;
     call AppendNewline();
    RPIncludeHasNL:
     call AppendCStr(&StrEndPLM);
     goto RPLoop;
    RPNotInclude:
     call RenderProseInline($line_start, $t);
    $has_beg =  call ContainsSliceCStr($line_start, $t, &StrBeginVerbOpen);
     goto RPLoop  if $has_beg  is  zero;
    $in_verb = 1;
     goto RPLoop;
    RPInVerb:
    $has_end =  call ContainsSliceCStr($line_start, $t, &StrEndVerbOpen);
     goto RPRaw  if $has_end  is  zero;
     # End marker found: leave verbatim, render this line as inline,
     # and re-check for a fresh begin marker on the same line so the
     # state transition matches the original two-test logic.
    $in_verb = 0;
     call RenderProseInline($line_start, $t);
    $has_beg =  call ContainsSliceCStr($line_start, $t, &StrBeginVerbOpen);
     goto RPLoop  if $has_beg  is  zero;
    $in_verb = 1;
     goto RPLoop;
    RPRaw:
     call AppendBytes($line_start, $t);
     goto RPLoop;
    RPDone:
     return;
    end
```

17. Rendering code bodies.

A code body is rendered inside the PLMCode fancyvrb environment with the following typographic conventions: language keywords bold, register names in roman italic, line comments in italic, hex literals and plain identifiers in the default monospace, and all non-identifier bytes escaped only for the fancyvrb commandchars. Chunk-reference sigils inside the body are expanded to `\plmref` calls.

IsPLMKeyword walks the packed keyword table and returns 1 if the given identifier slice matches any keyword.

⟨ ***Code body rendering*** ⟩ ≡

```
def IsPLMKeyword($ptr, $len) -> $r:
  register $h, $slot, $tmp, $stored, $kptr, $klen, $eq, $base;
 $r = 0;
  goto IPKDone  if $len  is  zero;


 $h =  call NameHash($ptr, $len);
 $slot = $h & KW_HASH_MASK;
 $base = &Keywords;
 IPKProbe:
 $tmp = $slot << 3;                  # * 8
 $tmp = $tmp + $KwHashTable;
 $stored = [$tmp].o;
```

```
    goto IPKDone  if $stored  is  zero;           # empty slot -> not a keyword
  $kptr = $base + $stored;
  $kptr = $kptr - 1;                          # points at the length byte
  $klen = [$kptr].bu;
  $tmp  = $klen - $len;
    goto IPKNext  if $tmp  is  nonzero;           # length mismatch, probe on
  $kptr = $kptr + 1;
  $eq   =  call StrEq($ptr, $len, $kptr, $klen);
    goto IPKFound  if $eq  is  nonzero;
  IPKNext:
  $slot = $slot + 1;
  $slot = $slot & KW_HASH_MASK;
    goto IPKProbe;
  IPKFound:
  $r = 1;
  IPKDone:
    return $r;
end
```

EmitChunkRef emits a chunk-reference hyperlink macro for a given normalized name.

⟨ ***Code body rendering*** ⟩ ≡

```
def EmitChunkRef($nptr, $nlen):
  call AppendCStr(&StrPlmref);
  call EmitChunkAnchor($nptr, $nlen);
  call AppendCStr(&StrBraceClose);
  call AppendCStr(&StrBraceOpen);
  call AppendTeXEscaped($nptr, $nlen);
  call AppendCStr(&StrBraceClose);
  return;
end
```

⟨ ***Code body rendering*** ⟩ ≡

```
def EmitLabelDefAnchor($nptr, $nlen):
  call AppendCStr(&StrRaisedHyperStart);
  call EmitLabelAnchor($nptr, $nlen);
  call AppendCStr(&StrRaisedHyperClose);
  call AppendBytes($nptr, $nlen);
  return;
end
```

⟨ ***Code body rendering*** ⟩ ≡

```
def EmitLabelCodeRef($nptr, $nlen):
  call AppendCStr(&StrHyperlink);
  call EmitLabelAnchor($nptr, $nlen);
  call AppendCStr(&StrBraceClose);
  call AppendCStr(&StrBraceOpen);
  call AppendBytes($nptr, $nlen);
  call AppendCStr(&StrBraceClose);
  return;
end
```

⟨ ***Code body rendering*** ⟩ ≡

```
def EmitProcedureDefAnchor($nptr, $nlen):
  call AppendCStr(&StrRaisedHyperStart);
  call EmitProcAnchor($nptr, $nlen);
  call AppendCStr(&StrRaisedHyperClose);
  call AppendBytes($nptr, $nlen);
  return;
end
```

⟨ ***Code body rendering*** ⟩ ≡

```
def EmitProcedureCodeRef($nptr, $nlen):
  call AppendCStr(&StrHyperlink);
  call EmitProcAnchor($nptr, $nlen);
  call AppendCStr(&StrBraceClose);
  call AppendCStr(&StrBraceOpen);
  call AppendBytes($nptr, $nlen);
  call AppendCStr(&StrBraceClose);
  return;
end
```

⟨ ***Code body rendering*** ⟩ ≡

```
def EmitConstantDefAnchor($nptr, $nlen):
  call AppendCStr(&StrRaisedHyperStart);
  call EmitConstAnchor($nptr, $nlen);
  call AppendCStr(&StrRaisedHyperClose);
  call AppendBytes($nptr, $nlen);
  return;
end
```

⟨ ***Code body rendering*** ⟩ ≡

```
def EmitConstantCodeRef($nptr, $nlen):
  call AppendCStr(&StrHyperlink);
  call EmitConstAnchor($nptr, $nlen);
  call AppendCStr(&StrBraceClose);
  call AppendCStr(&StrBraceOpen);
  call AppendBytes($nptr, $nlen);
  call AppendCStr(&StrBraceClose);
  return;
end
```

⟨ ***Code body rendering*** ⟩ ≡

```
def IsProcedureDefSite($bptr, $tok_start) -> $r:
  register $p, $q, $ch, $t;
 $r = 0;
 $p = $tok_start;


 IPDSkipSpace:
  goto IPDDone  if $p <=u $bptr;
 $q = $p - 1;
 $ch = [$q].bu;
 $t = $ch  cmpu 32;
  goto IPDCheckWord  if $t  is  pos;
 $p = $q;
  goto IPDSkipSpace;
 IPDCheckWord:
 $t = $p - $bptr;
 $t = $t - 3;
  goto IPDDone  if $t  is  neg;
 $q = $p - 3;
 $ch = [$q].bu;
 $t = $ch - 'd';
  goto IPDDone  if $t  is  nonzero;
 $ch = [$q + 1].bu;
 $t = $ch - 'e';
  goto IPDDone  if $t  is  nonzero;
 $ch = [$q + 2].bu;
 $t = $ch - 'f';
  goto IPDDone  if $t  is  nonzero;
  goto IPDYes  if $q <=u $bptr;
 $ch = [$q - 1].bu;
 $t =  call IsAlnumUS($ch);
  goto IPDDone  if $t  is  nonzero;
 IPDYes:
 $r = 1;
 IPDDone:
  return $r;
end
```

⟨ ***Code body rendering*** ⟩ ≡

```
def IsLabelDefSite($bptr, $tok_start, $tok_end, $end) -> $r:
  register $scan, $ch, $t;
 $r = 0;
 $t =  call IsTokenAtLineStart($bptr, $tok_start);
  goto ILDDone  if $t  is  zero;
 $scan = $tok_end;
 ILDSkipSpace:
  goto ILDDone  if $scan >=u $end;
 $ch = [$scan].bu;
 $t = $ch  cmpu 32;
  goto ILDCheckColon  if $t  is  pos;
 $scan = $scan + 1;
  goto ILDSkipSpace;
 ILDCheckColon:
 $t = $ch - ':';
  goto ILDDone  if $t  is  nonzero;
 $r = 1;
 ILDDone:
  return $r;
end
```

⟨ ***Code body rendering*** ⟩ ≡

```
def IsLabelRefSite($bptr, $tok_start) -> $r:
  register $p, $q, $ch, $t;
 $r = 0;
 $p = $tok_start;


 ILRSkipSpace:
  goto ILRDone  if $p <=u $bptr;
 $q = $p - 1;
 $ch = [$q].bu;
 $t = $ch  cmpu 32;
  goto ILRCheckPrev  if $t  is  pos;
 $p = $q;
  goto ILRSkipSpace;
 ILRCheckPrev:
 $t = $ch - '&';
  goto ILRYes  if $t  is  zero;
 $t = $p - $bptr;
 $t = $t - 4;
  goto ILRDone  if $t  is  neg;
 $q = $p - 4;
 $ch = [$q].bu;
 $t = $ch - 'g';
  goto ILRDone  if $t  is  nonzero;
 $ch = [$q + 1].bu;
 $t = $ch - 'o';
  goto ILRDone  if $t  is  nonzero;
 $ch = [$q + 2].bu;
 $t = $ch - 't';
  goto ILRDone  if $t  is  nonzero;
 $ch = [$q + 3].bu;
 $t = $ch - 'o';
  goto ILRDone  if $t  is  nonzero;
  goto ILRYes  if $q <=u $bptr;
 $ch = [$q - 1].bu;
 $t =  call IsAlnumUS($ch);
```

```
    goto ILRDone if $t is nonzero;
  ILRYes:
  $r = 1;
  ILRDone:
    return $r;
  end
```

RenderCodeBody is the main dispatcher for a raw body slice. It walks the slice byte by byte. At each position it first checks for an at-sigil (chunk reference or literal at-at), then for each of the five token categories used by the code renderer, falling through to a single-byte fallback for anything else.

⟨ ***Code body rendering*** ⟩ ≡

```
def RenderCodeBody($bptr, $blen):
  register $p, $end, $ch, $t;
  register $alpha, $digit;
  register $decl_kw, $const_kw;


  $p = $bptr;
  $end = $bptr + $blen;
  $decl_kw = 0;
  $const_kw = 0;
  RCBLoop:
    goto RCBDone if $p >=u $end;
  $ch = [$p].bu;
  $t = $ch - '@';
    goto RCBAt if $t is zero;
  $t = $ch - 34;
    goto RCBStr if $t is zero;
  $t = $ch - ' #';
    goto RCBHash if $t is zero;
  $t = $ch - '$';
    goto RCBReg if $t is zero;
  $t = $ch - '&';
    goto RCBAmp if $t is zero;
  $alpha = call IsAlphaUS($ch);
    goto RCBIdent if $alpha is nonzero;
  $digit = call IsDigit($ch);
    goto RCBDec if $digit is nonzero;
  $t = $ch - '\n';
    goto RCBPlainLineBreak if $t is zero;
  $t = $ch - ';';
    goto RCBPlainSemi if $t is zero;
    call AppendVerbatimEscaped($p, 1);
  $p = $p + 1;
    goto RCBLoop;
  RCBPlainLineBreak:
  $decl_kw = 0;
  $const_kw = 0;
    call AppendVerbatimEscaped($p, 1);
  $p = $p + 1;
    goto RCBLoop;
  RCBPlainSemi:
  $decl_kw = 0;
  $const_kw = 0;
    call AppendVerbatimEscaped($p, 1);
  $p = $p + 1;
    goto RCBLoop;
```

```
  RCBAt:
  ⟨RenderCodeBody: at-sigil⟩
   goto RCBLoop;
  RCBStr:
  ⟨RenderCodeBody: string literal⟩
   goto RCBLoop;
  RCBHash:
  ⟨RenderCodeBody: hash hex or comment⟩
   goto RCBLoop;
  RCBReg:
  ⟨RenderCodeBody: dollar register⟩
   goto RCBLoop;
  RCBAmp:
  ⟨RenderCodeBody: amp label ref⟩
   goto RCBLoop;
  RCBIdent:
  ⟨RenderCodeBody: identifier or keyword⟩
   goto RCBLoop;
  RCBDec:
  ⟨RenderCodeBody: decimal integer⟩
   goto RCBLoop;
  RCBDone:
   return;
 end
```

At-sigil bytes in a code body are either at-at (decoded to a single literal at-sign), or an at-angle-name-at-angle reference (expanded to a plmref macro), or a stray literal at-sign (emitted verbatim).

⟨***RenderCodeBody: at-sigil***⟩ ≡

```
 register $nxt, $tmp, $ns, $nr, $refp, $refl;
$tmp = $p + 1;
 goto RCBAtLit  if $tmp >=u $end;
$nxt = [$tmp].bu;


$tmp = $nxt - '@';
 goto RCBAtAt  if $tmp  is  zero;
$tmp = $nxt - '<';
 goto RCBAtAng  if $tmp  is  zero;
RCBAtLit:
 call AppendByte('@');
$p = $p + 1;
 goto RCBAtDone;
RCBAtAt:
 call AppendByte('@');
$p = $p + 2;
 goto RCBAtDone;
RCBAtAng:
$ns = $p + 2;
$nr = 0;
RCBAtScan:
$tmp = $ns + $nr;
 goto RCBAtLit  if $tmp >=u $end;
$ch = [$tmp].bu;
$tmp = $ch - '@';
 goto RCBAtInc  if $tmp  is  nonzero;
$tmp = $ns + $nr;
$tmp = $tmp + 1;
```

```
 goto RCBAtLit  if $tmp >=u $end;
$ch = [$tmp].bu;
$tmp = $ch - '>';
 goto RCBAtInc  if $tmp  is  nonzero;
$refp = $RefNameBuf;
$refl =  call NormNameTo($ns, $nr, $refp);
 call EmitChunkRef($refp, $refl);
$p = $ns + $nr;
$p = $p + 2;
 goto RCBAtDone;
RCBAtInc:
$nr = $nr + 1;
 goto RCBAtScan;
RCBAtDone:
```

A string literal is a double-quoted span with backslash escapes. The whole span (including the delimiters) is emitted via AppendVerbatimEscaped.

⟨ ***RenderCodeBody: string literal*** ⟩ ≡

```
 register $tok_start, $tok_end, $tmp;
$tok_start = $p;
$p = $p + 1;
RCBStrLoop:
 goto RCBStrEnd  if $p >=u $end;
$ch = [$p].bu;
$tmp = $ch - 92;
 goto RCBStrBS  if $tmp  is  zero;
$tmp = $ch - 34;
 goto RCBStrClose  if $tmp  is  zero;
$p = $p + 1;
 goto RCBStrLoop;
RCBStrBS:
$p = $p + 2;
 goto RCBStrLoop;
RCBStrClose:
$p = $p + 1;
RCBStrEnd:
$tok_end = $p;
$tmp = $tok_end - $tok_start;
 call AppendVerbatimEscaped($tok_start, $tmp);
```

A hash byte followed by a hex digit opens a hex literal (emitted verbatim); otherwise it opens a line comment that runs to end-of-line and is emitted in italic.

⟨ ***RenderCodeBody: hash hex or comment*** ⟩ ≡

```
 register $tok_start, $tok_end, $tmp;
$t = $p + 1;
 goto RCBCmt  if $t >=u $end;
$ch = [$t].bu;


$digit =  call IsHexDigit($ch);
 goto RCBCmt  if $digit  is  zero;
$tok_start = $p;
$p = $p + 1;
RCBHexLoop:
 goto RCBHexEnd  if $p >=u $end;
$ch = [$p].bu;
$digit =  call IsHexDigit($ch);
 goto RCBHexEnd  if $digit  is  zero;
$p = $p + 1;
 goto RCBHexLoop;
RCBHexEnd:
$tok_end = $p;
$tmp = $tok_end - $tok_start;
 call AppendBytes($tok_start, $tmp);
 goto RCBHashDone;
RCBCmt:
$tok_start = $p;
RCBCmtLoop:
 goto RCBCmtEnd  if $p >=u $end;
$ch = [$p].bu;
$tmp = $ch - '\n';
 goto RCBCmtEnd  if $tmp  is  zero;
$p = $p + 1;
 goto RCBCmtLoop;
RCBCmtEnd:
$tok_end = $p;
$tmp = $tok_end - $tok_start;
 call AppendCStr(&StrItshape);
 call AppendVerbatimEscaped($tok_start, $tmp);
 call AppendCStr(&StrBraceClose);
RCBHashDone:
```

A dollar sign that introduces a register is followed by an identifier or a digit run. The emitted text uses a normalfont-italic span. A bare dollar with nothing suitable after falls through to the single-byte escape.

⟨ ***RenderCodeBody: dollar register*** ⟩ ≡

```
 register $tok_start, $tok_end, $tmp;
$t = $p + 1;
 goto RCBRegFallback  if $t >=u $end;
$ch = [$t].bu;


$alpha =  call IsAlphaUS($ch);
 goto RCBRegId  if $alpha  is  nonzero;
$digit =  call IsDigit($ch);
 goto RCBRegNum  if $digit  is  nonzero;
RCBRegFallback:
 call AppendVerbatimEscaped($p, 1);
$p = $p + 1;
 goto RCBRegDone;
```

```
RCBRegId:
$tok_start = $p + 1;
$p = $p + 2;
RCBRegIdLoop:
 goto RCBRegIdEnd  if $p >=u $end;
$ch = [$p].bu;
$alpha =  call IsAlnumUS($ch);
 goto RCBRegIdEnd  if $alpha  is  zero;
$p = $p + 1;
 goto RCBRegIdLoop;
RCBRegIdEnd:
$tok_end = $p;
$tmp = $tok_end - $tok_start;
 call AppendCStr(&StrNormItReg);
 call AppendBytes($tok_start, $tmp);
 call AppendCStr(&StrBraceClose);
 goto RCBRegDone;
RCBRegNum:
$tok_start = $p + 1;
$p = $p + 2;
RCBRegNumLoop:
 goto RCBRegNumEnd  if $p >=u $end;
$ch = [$p].bu;
$digit =  call IsDigit($ch);
 goto RCBRegNumEnd  if $digit  is  zero;
$p = $p + 1;
 goto RCBRegNumLoop;
RCBRegNumEnd:
$tok_end = $p;
$tmp = $tok_end - $tok_start;
 call AppendCStr(&StrNormItReg);
 call AppendBytes($tok_start, $tmp);
 call AppendCStr(&StrBraceClose);
RCBRegDone:
```

An ampersand followed by an identifier is rendered as an address-of reference and links to that identifier's label anchor.

⟨ ***RenderCodeBody: amp label ref*** ⟩ ≡

```
 register $tok_start, $tok_end, $tmp;
$t = $p + 1;
 goto RCBAmpFallback  if $t >=u $end;
$ch = [$t].bu;
$alpha =  call IsAlphaUS($ch);
 goto RCBAmpFallback  if $alpha  is  zero;
 call AppendVerbatimEscaped($p, 1);
$tok_start = $p + 1;
$p = $p + 2;
RCBAmpLoop:
 goto RCBAmpEnd  if $p >=u $end;
$ch = [$p].bu;
$alpha =  call IsAlnumUS($ch);
 goto RCBAmpEnd  if $alpha  is  zero;
$p = $p + 1;
 goto RCBAmpLoop;
RCBAmpEnd:
$tok_end = $p;
$tmp = $tok_end - $tok_start;
 call EmitLabelCodeRef($tok_start, $tmp);
```

```
    goto RCBAmpDone;
RCBAmpFallback:
    call AppendVerbatimEscaped($p, 1);
$p = $p + 1;
RCBAmpDone:
```

An identifier run may be a constant definition/reference, a label definition/reference, a procedure definition/call target, or a plain identifier.

⟨ ***RenderCodeBody: identifier or keyword*** ⟩ ≡

```
    register $tok_start, $tok_end, $tmp, $scan, $nxt;
$tok_start = $p;
$p = $p + 1;
RCBIdLoop:

    goto RCBIdEnd  if $p >=u $end;
$ch = [$p].bu;
$alpha =  call IsAlnumUS($ch);
    goto RCBIdEnd  if $alpha  is  zero;
$p = $p + 1;
    goto RCBIdLoop;
RCBIdEnd:
$tok_end = $p;
$tmp = $tok_end - $tok_start;
$t = $const_kw;
    goto RCBIdCheckConstKeyword  if $t  is  zero;
    call EmitConstantDefAnchor($tok_start, $tmp);
$const_kw = 0;
    goto RCBIdDone;
RCBIdCheckConstKeyword:
$t =  call IsConstKeyword($tok_start, $tmp);
    goto RCBIdCheckDataLabel  if $t  is  zero;
$const_kw = 1;
    goto RCBIdPlainText;
RCBIdCheckDataLabel:
$t = $decl_kw;
    goto RCBIdCheckLabelDef  if $t  is  zero;
    call EmitLabelDefAnchor($tok_start, $tmp);
$decl_kw = 0;
    goto RCBIdDone;
RCBIdCheckLabelDef:
$t =  call IsLabelDefSite($bptr, $tok_start, $tok_end, $end);
    goto RCBIdCheckDataLabelDef  if $t  is  zero;
    call EmitLabelDefAnchor($tok_start, $tmp);
$decl_kw = 0;
    goto RCBIdDone;
RCBIdCheckDataLabelDef:
$t =  call IsDataDeclKeyword($tok_start, $tmp);
    goto RCBIdTryKeyword  if $t  is  zero;
$decl_kw = 1;
    goto RCBIdPlainText;
RCBIdTryKeyword:
$t =  call IsPLMKeyword($tok_start, $tmp);
    goto RCBIdTryProc  if $t  is  zero;
    call AppendCStr(&StrBfseries);
    call AppendBytes($tok_start, $tmp);
    call AppendCStr(&StrBraceClose);
    goto RCBIdDone;
```

```
RCBIdTryProc:
$scan = $tok_end;
RCBIdSkipSpace:
 goto RCBIdPlain  if $scan >=u $end;
$nxt = [$scan].bu;
$t = $nxt  cmpu 32;
 goto RCBIdCheckCall  if $t  is  pos;
$scan = $scan + 1;
 goto RCBIdSkipSpace;
RCBIdCheckCall:
$t = $nxt - '(';
 goto RCBIdPlain  if $t  is  nonzero;
$t =  call IsProcedureName($tok_start, $tmp);
 goto RCBIdPlain  if $t  is  zero;
$t =  call IsProcedureDefSite($bptr, $tok_start);
 goto RCBIdUse  if $t  is  zero;
 call EmitProcedureDefAnchor($tok_start, $tmp);
 goto RCBIdDone;
RCBIdUse:
 call EmitProcedureCodeRef($tok_start, $tmp);
 goto RCBIdDone;
RCBIdPlainLabel:
 call EmitLabelCodeRef($tok_start, $tmp);
 goto RCBIdDone;
RCBIdPlain:
$t =  call IsLabelName($tok_start, $tmp);
 goto RCBIdCheckConstRef  if $t  is  zero;
 goto RCBIdPlainLabel;
RCBIdCheckConstRef:
$t =  call IsConstantName($tok_start, $tmp);
 goto RCBIdPlainText  if $t  is  zero;
 $scan = $tok_end;
 RCBIdConstGap:
  goto RCBIdConstUse  if $scan >=u $end;
 $nxt = [$scan].bu;
 $t = $nxt  cmpu 32;
  goto RCBIdConstEq  if $t  is  pos;
 $scan = $scan + 1;
  goto RCBIdConstGap;
 RCBIdConstEq:
 $t = $nxt - '=';
  goto RCBIdConstUse  if $t  is  nonzero;
  call EmitConstantDefAnchor($tok_start, $tmp);
  goto RCBIdDone;
 RCBIdConstUse:
 call EmitConstantCodeRef($tok_start, $tmp);
 goto RCBIdDone;
RCBIdPlainText:
$t =  call IsPLMKeyword($tok_start, $tmp);
 goto RCBIdReallyPlain  if $t  is  zero;
 call AppendCStr(&StrBfseries);
 call AppendBytes($tok_start, $tmp);
 call AppendCStr(&StrBraceClose);
 goto RCBIdDone;
RCBIdReallyPlain:
 call AppendBytes($tok_start, $tmp);
RCBIdDone:
```

A decimal integer run is emitted verbatim.

⟨ ***RenderCodeBody: decimal integer*** ⟩ ≡

```
 register $tok_start, $tok_end, $tmp;
$tok_start = $p;
$p = $p + 1;
RCBDecLoop:
 goto RCBDecEnd  if $p >=u $end;
$ch = [$p].bu;
$digit =  call IsDigit($ch);
 goto RCBDecEnd  if $digit  is  zero;
$p = $p + 1;
 goto RCBDecLoop;
RCBDecEnd:
$tok_end = $p;
$tmp = $tok_end - $tok_start;
 call AppendBytes($tok_start, $tmp);
```

18. Emitting the document.

EmitDocument ties everything together. It writes the preamble, the title stanza, and the `\begin{document}` boilerplate; iterates over the parsed section table emitting prose, major headings, and code chunks in order; and finally emits the cross-reference index and `\end{document}`.

⟨ ***Document emission*** ⟩ ≡

```
def EmitPreamble():
  register $name_len;
  call AppendCStr(&TexPreamble);
  call AppendCStr(&StrTitleOpen);
 $name_len =  call StrLen($SourceNameBuf);
  call AppendTeXEscaped($SourceNameBuf, $name_len);
  call AppendCStr(&StrTitleClose);
  return;
end
```

EmitSectionPageGuard reserves enough vertical space for a major section heading and a few lines of following prose. This prevents a section title from landing alone at the bottom of a page.

⟨ ***Document emission*** ⟩ ≡

```
def EmitSectionPageGuard():
  call AppendCStr(&StrNeedSpaceOpen);
  call AppendInt(SECTION_NEEDSPACE_LINES);
  call AppendCStr(&StrNeedSpaceClose);
  return;
end
```

EmitSectionHeading emits a major section heading with its hypertarget and display text. We emit a starred section plus an explicit table-of-contents entry so the visible heading shows the computed major-section number inline as “N. Title”, without LaTeX adding a separate automatic section number.

⟨ ***Document emission*** ⟩ ≡

```
def EmitSectionHeading($num, $hptr, $hlen):
  call AppendCStr(&StrHyperStart);
  call EmitSecAnchor($num);
  call AppendCStr(&StrHyperMid);
  call AppendNewline();
  call AppendCStr(&StrSectionStar);
  call AppendInt($num);
  call AppendCStr(&StrDotSpace);
  call AppendTeXEscaped($hptr, $hlen);
  call AppendCStr(&StrBraceClose);
  call AppendNewline();
  call AppendCStr(&StrAddTocSection);
  call AppendInt($num);
  call AppendCStr(&StrDotSpace);
  call AppendTeXEscaped($hptr, $hlen);
  call AppendCStr(&StrBraceClose);
  call AppendNewline();
  call AppendNewline();
  return;
end
```

CountLinesInSlice returns the number of logical lines in a byte slice by counting linefeed bytes and adding one when the slice does not already end with a linefeed. Empty slices count as zero lines.

⟨ ***Document emission*** ⟩ ≡

```
def CountLinesInSlice($ptr, $len) -> $count:
  register $p, $end, $ch, $last_was_nl, $tmp;
 $count = 0;
  goto CLSDone  if $len  is  zero;
 $p = $ptr;
 $end = $ptr + $len;
 $last_was_nl = 0;
 CLSLoop:
  goto CLSFinish  if $p >=u $end;
 $ch = [$p].bu;
 $tmp = $ch - '\n';
  goto CLSNotNL  if $tmp  is  nonzero;
 $count = $count + 1;
 $last_was_nl = 1;
  goto CLSNext;
 CLSNotNL:
 $last_was_nl = 0;
 CLSNext:
 $p = $p + 1;
  goto CLSLoop;
 CLSFinish:
  goto CLSDone  if $last_was_nl  is  nonzero;
 $count = $count + 1;
 CLSDone:
  return $count;
end
```

PrefixLenForLines returns the byte length of the shortest prefix that contains the requested number of logical lines, stopping early at end of slice. It is used to keep the start of a long chunk

with its heading, then continue the remainder as a second verbatim block if needed.

⟨***Document emission***⟩ ≡

```
def PrefixLenForLines($ptr, $len, $want_lines) -> $prefix_len:
  register $p, $end, $ch, $lines, $tmp;
 $prefix_len = 0;
  goto PLFLDone  if $len  is  zero;
  goto PLFLDone  if $want_lines  is  zero;
 $p = $ptr;
 $end = $ptr + $len;
 $lines = 0;
 PLFLLoop:
  goto PLFLDone  if $p >=u $end;
 $ch = [$p].bu;
 $p = $p + 1;
 $prefix_len = $prefix_len + 1;
 $tmp = $ch - '\n';
  goto PLFLLoop  if $tmp  is  nonzero;
 $lines = $lines + 1;
 $tmp = $lines - $want_lines;
  goto PLFLDone  if $tmp  is  zero;
  goto PLFLLoop;
 PLFLDone:
  return $prefix_len;
end
```

Reserve enough vertical space for a chunk heading plus a few body lines before starting a code block. This prevents a heading line or opening equivalence marker from being stranded at the foot of a page.

⟨***Document emission***⟩ ≡

```
def EmitChunkPageGuard($body_lines):
  register $lines, $tmp;
 $lines = $body_lines + 2;
 $tmp = $lines - CHUNK_NEEDSPACE_LINES;
  goto ECPGEmit  if $tmp  is  nonpos;
 $lines = CHUNK_NEEDSPACE_LINES;
 ECPGEmit:
  call AppendCStr(&StrNeedSpaceOpen);
  call AppendInt($lines);
  call AppendCStr(&StrNeedSpaceClose);
  return;
end
```

EmitChunkHeading emits the heading line for one code chunk. A first definition uses the equivalence sigil; continuation definitions use the plus-equivalence sigil.

⟨ ***Document emission*** ⟩ ≡

```
def EmitChunkHeading($nptr, $nlen, $defn_count):
  register $t;
  call AppendCStr(&StrMedSkip);
 $t = $defn_count - 1;
  goto ECHContinuation  if $t  is  nonzero;
  call AppendCStr(&StrHyperStart);
  call EmitChunkAnchor($nptr, $nlen);
  call AppendCStr(&StrHyperMid);
  call AppendCStr(&StrBfChunkName);
  call AppendTeXEscaped($nptr, $nlen);
  call AppendCStr(&StrBraceClose);
  call AppendCStr(&StrChunkDefEq);
  goto ECHNoPage;
 ECHContinuation:
  call AppendCStr(&StrBfChunkName);
  call AppendTeXEscaped($nptr, $nlen);
  call AppendCStr(&StrChunkContEq);
 ECHNoPage:
  call AppendCStr(&StrNoPageBreak);
  return;
end
```

EmitSections walks the section table in order. A prose section with a heading is emitted as both a sectioning command and, if it has body text, the body is rendered. A code chunk emits its heading plus a PLMCode block.

⟨ ***Document emission*** ⟩ ≡

```
def EmitSections():
  register $i, $entry, $kind, $hptr, $hlen, $tptr, $tlen, $num, $cur;
  register $defn_count, $body_lines, $samepage, $prefix_len, $rest_ptr, $rest_len, $tmp;
 $cur = 0;


 $i = 0;
 ESLoop:
  goto ESDone  if $i >=u $sec_cnt;
 $entry =  call SectionEntryAt($i);
 $kind = [$entry].o;
 $hptr = [$entry + SEC_HPTR].o;
 $hlen = [$entry + SEC_HLEN].o;
 $tptr = [$entry + SEC_TPTR].o;
 $tlen = [$entry + SEC_TLEN].o;
 $num  = [$entry + SEC_NUM].o;
  goto ESCode  if $kind  is  nonzero;
  goto ESMinor  if $num  is  zero;
 $cur = $num;
  call EmitSectionPageGuard();
  call EmitSectionHeading($num, $hptr, $hlen);
  goto ESProse;
 ESMinor:
  goto ESNext  if $tlen  is  zero;
 ESProse:
  call RenderProse($tptr, $tlen);
  call AppendNewline();
  call AppendNewline();
  goto ESNext;
 ESCode:
```

```
  $defn_count = $num;
   goto ESHaveDefn  if $defn_count  is  nonzero;
  $defn_count = 1;
  ESHaveDefn:
  $body_lines =  call CountLinesInSlice($tptr, $tlen);
   call EmitChunkPageGuard($body_lines);
  $samepage = $body_lines - CHUNK_SAMEPAGE_MAX;
   goto ESLongChunk  if $samepage  is  pos;
   call AppendCStr(&StrBeginSamePage);
   call EmitChunkHeading($hptr, $hlen, $defn_count);
   call AppendCStr(&StrBeginPLMSamePage);
   goto ESRenderWhole;
  ESLongChunk:
  $tmp = $body_lines - CHUNK_HEADKEEP_LINES;
   goto ESNormalPLM  if $tmp  is  nonpos;
  $prefix_len =  call PrefixLenForLines($tptr, $tlen, CHUNK_HEADKEEP_LINES);
   call AppendCStr(&StrBeginSamePage);
   call EmitChunkHeading($hptr, $hlen, $defn_count);
   call AppendCStr(&StrBeginPLMSamePage);
   call RenderCodeBody($tptr, $prefix_len);
   call AppendNewline();
   call AppendCStr(&StrEndPLMCont);
   call AppendCStr(&StrEndSamePage);
  $rest_ptr = $tptr + $prefix_len;
  $rest_len = $tlen - $prefix_len;
   goto ESNext  if $rest_len  is  zero;
   call AppendCStr(&StrBeginPLM);
   call RenderCodeBody($rest_ptr, $rest_len);
   call AppendNewline();
   call AppendCStr(&StrEndPLM);
   goto ESNext;
  ESNormalPLM:
   call AppendCStr(&StrBeginSamePage);
   call EmitChunkHeading($hptr, $hlen, $defn_count);
   call AppendCStr(&StrBeginPLM);
  ESRenderWhole:
   call RenderCodeBody($tptr, $tlen);
   call AppendNewline();
   call AppendCStr(&StrEndPLM);
   call AppendCStr(&StrEndSamePage);
  ESNext:
  $i = $i + 1;
   goto ESLoop;
  ESDone:
   return;
 end
```

⟨***Document emission***⟩ ≡

```
 def EmitDocument():
   call EmitPreamble();
   call EmitSections();
   call AppendCStr(&StrEndDoc);
   return;
 end
```

19. Cross-reference index.

The cross-reference index is a single section star block holding four subsection star blocks (Chunks, Labels, Constants, Registers). Each block is a description list, one item per unique name or register. The chunk list reports defined-in and used-in section ranges; the label and constant lists report a defining section plus used-in; the register list reports a flat used-in list.

⟨ ***Cross-reference scans*** ⟩ ≡

```
def EmitSectionListForChunkDefs($entry):
  register $base, $len, $i, $poolp, $sec, $first;
 $base = [$entry + UNIQ_DEFS].o;
 $len  = [$entry + UNIQ_DLEN].o;
 $i = 0;
 $first = 1;
 ESLCDLoop:
  goto ESLCDDone  if $i >=u $len;
 $poolp = $base + $i;
 $poolp = $poolp * 8;
 $poolp = $poolp + $SecRefPool;
 $sec = [$poolp].o;
  goto ESLCDEmit  if $first  is  nonzero;
  call AppendCStr(&StrCommaSpace);
 ESLCDEmit:
  call EmitSecLink($sec);
 $first = 0;
 $i = $i + 1;
  goto ESLCDLoop;
 ESLCDDone:
  return;
end
```

⟨ ***Cross-reference scans*** ⟩ ≡

```
def EmitSectionListForChunkUses($entry):
  register $base, $len, $i, $poolp, $sec, $first;
 $base = [$entry + UNIQ_USES].o;
 $len  = [$entry + UNIQ_ULEN].o;
  goto ESLCUHave  if $len  is  nonzero;
  call AppendCStr(&StrDashDash);
  return;
 ESLCUHave:
 $i = 0;
 $first = 1;
 ESLCULoop:
  goto ESLCUDone  if $i >=u $len;
 $poolp = $base + $i;
 $poolp = $poolp * 8;
 $poolp = $poolp + $SecRefPool;
 $sec = [$poolp].o;
  goto ESLCUEmit  if $first  is  nonzero;
  call AppendCStr(&StrCommaSpace);
 ESLCUEmit:
  call EmitSecLink($sec);
 $first = 0;
 $i = $i + 1;
  goto ESLCULoop;
 ESLCUDone:
  return;
```

```
end
```

⟨ ***Cross-reference scans*** ⟩ ≡

```
def EmitLabelUseList($entry):
  register $base, $len, $i, $poolp, $sec, $first;
 $base = [$entry + LBL_USES].o;
 $len  = [$entry + LBL_ULEN].o;
  goto ELULHave  if $len  is  nonzero;
  call AppendCStr(&StrDashDash);
  return;
 ELULHave:
 $i = 0;
 $first = 1;
 ELULLoop:
  goto ELULDone  if $i >=u $len;
 $poolp = $base + $i;
 $poolp = $poolp * 8;
 $poolp = $poolp + $SecRefPool;
 $sec = [$poolp].o;
  goto ELULEmit  if $first  is  nonzero;
  call AppendCStr(&StrCommaSpace);
 ELULEmit:
  call EmitSecLink($sec);
 $first = 0;
 $i = $i + 1;
  goto ELULLoop;
 ELULDone:
  return;
end
```

⟨ ***Cross-reference scans*** ⟩ ≡

```
def EmitConstantUseList($entry):
  register $base, $len, $i, $poolp, $sec, $first;
 $base = [$entry + CONST_USES].o;
 $len  = [$entry + CONST_ULEN].o;
  goto ECULHave  if $len  is  nonzero;
  call AppendCStr(&StrDashDash);
  return;
 ECULHave:
 $i = 0;
 $first = 1;
 ECULLoop:
  goto ECULDone  if $i >=u $len;
 $poolp = $base + $i;
 $poolp = $poolp * 8;
 $poolp = $poolp + $SecRefPool;
 $sec = [$poolp].o;
  goto ECULEmit  if $first  is  nonzero;
  call AppendCStr(&StrCommaSpace);
 ECULEmit:
  call EmitSecLink($sec);
 $first = 0;
 $i = $i + 1;
  goto ECULLoop;
 ECULDone:
  return;
```

```
end
```

⟨ ***Cross-reference scans*** ⟩ ≡

```
def EmitChunkIndex():
  register $i, $entry, $nptr, $nlen;
  goto ECIDone  if $uc_cnt  is  zero;
  call AppendCStr(&StrSubChunks);

  call AppendCStr(&StrBeginDesc);
 $i = 0;
 ECILoop:
  goto ECIEnd  if $i >=u $uc_cnt;
 $entry = $i * UNIQ_ENTRY_SIZE;
 $entry = $entry + $UniqueChunks;
 $nptr = [$entry].o;
 $nlen = [$entry + 8].o;
  call AppendCStr(&StrItemOpen);
  call AppendCStr(&StrHyperlink);
  call EmitChunkAnchor($nptr, $nlen);
  call AppendCStr(&StrBraceClose);
  call AppendCStr(&StrBraceOpen);
  call AppendCStr(&StrBfChunkName);
  call AppendTeXEscaped($nptr, $nlen);
  call AppendCStr(&StrBraceClose);
  call AppendCStr(&StrBraceClose);
  call AppendCStr(&StrItemClose);
  call AppendCStr(&StrDefinedIn);
  call EmitSectionListForChunkDefs($entry);
  call AppendCStr(&StrUsedIn);
  call EmitSectionListForChunkUses($entry);
  call AppendNewline();
 $i = $i + 1;
  goto ECILoop;
 ECIEnd:
  call AppendCStr(&StrEndDesc);
 ECIDone:
  return;
end
```

⟨ ***Cross-reference scans*** ⟩ ≡

```
def EmitLabelIndex():
  register $i, $entry, $nptr, $nlen, $def;
  goto ELIDone  if $lbl_cnt  is  zero;
  call AppendCStr(&StrSubLabels);

  call AppendCStr(&StrBeginDesc);
 $i = 0;
 ELILoop:
  goto ELIEnd  if $i >=u $lbl_cnt;
 $entry = $i * LBL_ENTRY_SIZE;
 $entry = $entry + $LabelTable;
 $nptr = [$entry].o;
 $nlen = [$entry + 8].o;
 $def  = [$entry + LBL_DEF].o;
  call AppendCStr(&StrItemOpen);
```

```
    call AppendCStr(&StrTexttt);
    call AppendTeXEscaped($nptr, $nlen);
    call AppendCStr(&StrBraceClose);
    call AppendCStr(&StrItemClose);
    call AppendCStr(&StrDefinedIn);
    call EmitSecLink($def);
    call AppendCStr(&StrDefMarker);
    call AppendCStr(&StrUsedIn);
    call EmitLabelUseList($entry);
    call AppendNewline();
  $i = $i + 1;
    goto ELILoop;
  ELIEnd:
    call AppendCStr(&StrEndDesc);
  ELIDone:
    return;
  end
```

⟨ ***Cross-reference scans*** ⟩ ≡

```
  def EmitConstantIndex():
    register $i, $entry, $nptr, $nlen, $def;
    goto ECONDone  if $const_cnt  is  zero;
    call AppendCStr(&StrSubConstants);

    call AppendCStr(&StrBeginDesc);
  $i = 0;
  ECONLoop:
    goto ECONEnd  if $i >=u $const_cnt;
  $entry = $i * CONST_ENTRY_SIZE;
  $entry = $entry + $ConstantTable;
  $nptr = [$entry + CONST_NPTR].o;
  $nlen = [$entry + CONST_NLEN].o;
  $def  = [$entry + CONST_DEF].o;
    call AppendCStr(&StrItemOpen);
    call AppendCStr(&StrHyperlink);
    call EmitConstAnchor($nptr, $nlen);
    call AppendCStr(&StrBraceClose);
    call AppendCStr(&StrBraceOpen);
    call AppendCStr(&StrTexttt);
    call AppendTeXEscaped($nptr, $nlen);
    call AppendCStr(&StrBraceClose);
    call AppendCStr(&StrBraceClose);
    call AppendCStr(&StrItemClose);
    call AppendCStr(&StrDefinedIn);
    goto ECONDefMissing  if $def  is  zero;
    call EmitSecLink($def);
    call AppendCStr(&StrDefMarker);
    goto ECONUses;
  ECONDefMissing:
    call AppendCStr(&StrDashDash);
  ECONUses:
    call AppendCStr(&StrUsedIn);
    call EmitConstantUseList($entry);
    call AppendNewline();
  $i = $i + 1;
    goto ECONLoop;
  ECONEnd:
    call AppendCStr(&StrEndDesc);
```

```
 ECONDone:
  return;
end
```

⟨ ***Cross-reference scans*** ⟩ ≡

```
def SortRegisters():
  register $i, $j, $pj, $pp, $key, $a, $b, $t, $tmp;
 $key = $SortKeyBuf;
 $i = 1;


 SRLoop:
 $t = $i  cmpu $reg_cnt;
  goto SRDone  if $t  is  nonneg;
 $pj = $i * REG_ENTRY_SIZE;
 $pj = $pj + $RegisterTable;
  call CopyEntry($key, $pj, REG_ENTRY_SIZE);
 $b = [$key + REG_NUM].o;
 $j = $i;
 SRInner:
  goto SRPlace  if $j  is  zero;
 $tmp = $j - 1;
 $pp = $tmp * REG_ENTRY_SIZE;
 $pp = $pp + $RegisterTable;
 $a = [$pp + REG_NUM].o;
 $t = $a - $b;
  goto SRPlace  if $t  is  nonpos;
 $pj = $j * REG_ENTRY_SIZE;
 $pj = $pj + $RegisterTable;
  call CopyEntry($pj, $pp, REG_ENTRY_SIZE);
 $j = $j - 1;
  goto SRInner;
 SRPlace:
 $pj = $j * REG_ENTRY_SIZE;
 $pj = $pj + $RegisterTable;
  call CopyEntry($pj, $key, REG_ENTRY_SIZE);
 $i = $i + 1;
  goto SRLoop;
 SRDone:
  return;
end
```

⟨ ***Cross-reference scans*** ⟩ ≡

```
def EmitRegisterIndex():
  register $i, $entry, $num, $base, $len, $j, $poolp, $sec, $first;
  goto ERIDone  if $reg_cnt  is  zero;
  call SortRegisters();


  call AppendCStr(&StrSubRegisters);
  call AppendCStr(&StrBeginDesc);
 $i = 0;
 ERILoop:
  goto ERIEnd  if $i >=u $reg_cnt;
 $entry = $i * REG_ENTRY_SIZE;
 $entry = $entry + $RegisterTable;
 $num  = [$entry + REG_NUM].o;
 $base = [$entry + REG_USES].o;
```

```
    $len  = [$entry + REG_ULEN].o;
     call AppendCStr(&StrItemOpen);
     call AppendCStr(&StrTexttt);
     call AppendCStr(&StrDollar);
     call AppendInt($num);
     call AppendCStr(&StrBraceClose);
     call AppendCStr(&StrItemClose);
     call AppendByte(' ');
    $j = 0;
    $first = 1;
    ERIULoop:
     goto ERIUDone  if $j >=u $len;
    $poolp = $base + $j;
    $poolp = $poolp * 8;
    $poolp = $poolp + $SecRefPool;
    $sec = [$poolp].o;
     goto ERIUEmit  if $first  is  nonzero;
     call AppendCStr(&StrCommaSpace);
    ERIUEmit:
     call EmitSecLink($sec);
    $first = 0;
    $j = $j + 1;
     goto ERIULoop;
    ERIUDone:
     call AppendNewline();
    $i = $i + 1;
     goto ERILoop;
    ERIEnd:
     call AppendCStr(&StrEndDesc);
    ERIDone:
     return;
  end
```

⟨ ***Cross-reference scans*** ⟩ ≡

```
  def EmitCrossRefIndex():
    call AppendCStr(&StrHrule);
    call AppendCStr(&StrXrefHead);
    call EmitChunkIndex();
    call EmitLabelIndex();
    call EmitConstantIndex();
    call EmitRegisterIndex();
    return;
  end
```

20. Error handling.

All errors here are fatal. A partially written document is worse than none at all: it might be tangled through a build script and produce broken output silently. ErrHalt writes the message to standard error via FPUTS and halts with exit status 1.

⟨ ***Error handler*** ⟩ ≡

```
def ErrHalt($msg):
 $out = $msg;
  trap 0, FPUTS, STDERR;
  trap 0, HALT, 1;
end
```

21. Showing the work.

When weaving is complete, we output. The byte count in `$out_end` is flushed via a single FWRITE trap.

⟨ ***Write output to stdout*** ⟩ ≡

```
$out = $FWriteArgs;
[$FWriteArgs].o      = $OutputBuf;
[$FWriteArgs + 8].o = $out_end;
 trap 0, FWRITE, STDOUT;
```

E Bootstrap Compiler: Woven Output

"My mind is bent to tell of bodies changed into new forms."
— Ovid, *Metamorphoses*, I.1–2 (c. 8 AD); transl. A. S. Kline (2000)

This appendix shows the complete woven output of the bootstrap compiler source `plmmix-compiler.plw`.

1. Introduction.

This program is `plmmix-compiler`: the PL/MMIX compiler, written as a PL/MMIX literate program so that it can compile itself. It reads a `.plm` source from standard input (the flat source produced by `plmmix-tangle`) and writes MMIXAL assembly to standard output. The resulting `.mms` file is fed to `mmixal` to produce a loadable MMIX object file.

Role in the toolchain

The full build pipeline for any PL/MMIX program `foo.plw` is:

```
plmmix-tangle   foo.plw   > foo.plm       % strip literate markup
plmmix-compiler foo.plm   > foo.mms       % PL/MMIX -> MMIXAL
mmixal foo.mms                            % MMIXAL -> foo.mmo
mmix   foo.mmo                            % execute
```

Only the second step is implemented here. The first step is already implemented by `plmmix-tangle.plw`; the last two are external tools.

Pipeline inside the compiler

Compilation proceeds in four phases, each consuming the output of the previous one through a shared in-memory arena:

Read. The entire `.plm` input is slurped into `SourceBuf` with a single `FREAD`. After this phase the source is treated as immutable; all later phases reference slices of it by pointer.

Lex. The source is scanned once left-to-right into a flat array of fixed-size token records in `TokenBuf`. Each token records its kind, its literal value (decoded integer, or a pointer and length into `SourceBuf`/`StringPool`), and its line and column for error messages.

Parse. A recursive-descent parser consumes the token stream and builds an abstract syntax tree in `NodeBuf`. AST nodes are fixed-size records; nested children are referenced by offset so that the tree is a flat array rather than a linked structure.

Generate. The AST is walked once; each node emits one or more lines of MMIXAL text into `OutputBuf`. A single `FWRITE` flushes the buffer at the end of `main`.

Design constraints

Like `plmmix-tangle`, this program runs on bare MMIX with no operating system allocator. Every buffer is fixed size, declared at load time in the data-segment section. A hard limit breach prints a clear message to standard error and halts with status 1; no partial output is ever emitted.

The compiler is written using only the features it can itself compile, so that once bootstrapped it can recompile its own source without needing any external compiler implementation.

Language features deliberately avoided in this source

A few PL/MMIX constructs that the language otherwise supports are avoided throughout this file so that the bootstrap path stays simple and so that hand-written test fixtures can match the generated output exactly. These are coding-style choices for this source, not language restrictions:

- `call` inside expression positions (for example as the right-hand side of a binary operator or inside another call's argument list). Every call is written as a top-level statement; the result is captured into a temporary first.
- Combined arithmetic-and-address forms in a single operand, such as `base + idx * stride` inside a load or store; the effective address is always built into a temporary register first.

Multi-value return assignment of the form `$a, $b = call F(...)` is a supported language feature (parsed as `NK_CALLASN` in the statement parser and lowered to a multi-register `POP` in the procedure-body generator); it simply happens not to be used in this source.

A note on coding style

Many procedures here use named labels and `goto $X$ if cond` rather than the structured `if`/`while`/`end` forms that PL/MMIX also supports. This is a deliberate parity choice with the original bootstrap source: the same control-flow shape compiles identically through every revision of the toolchain, so a regression in code generation is easy to bisect against earlier compiler versions. The two styles are equivalent; structured forms are used freely in the newer top-level dispatch code.

How this file is organised

The sections that follow are grouped by phase and by layer:

- The constants and data-segment sections declare the fixed arenas and tables that every phase relies on.
- The main-program section gives the top-level program structure.
- The reading-the-source section covers `ReadAll` and `InitSourceName`.
- The string, integer-conversion, and output-helper sections provide the shared byte-level utilities.
- The error-handling section is the error reporter.
- The token-kinds, lexer, and parser sections define the front end.
- The generator sections define the top-level AST walk, the driver banner, procedure-body emission, and the control-flow helpers.

The processing order of the code mirrors the order of the narrative: read, lex, parse, generate, write. The tangled output order is explicit in the main-program skeleton: constants and the data segment come first, followed by the main program stub and the procedure collections it calls.

2. Constants and limits.

All buffer sizes are fixed at compile time. They are chosen to be generous for self-hosting (the compiler must fit its own source plus all intermediate arrays) while remaining small enough to

load quickly on the simulator. Each limit can be raised without further code changes up to the limits of the MMIX address space.

The `@<Constants@>=` chunk is opened here and re-opened many times throughout the file, in every section that introduces its own private constants (token kinds, error message offsets, generator opcodes, etc.). The accumulated definitions are tangled together in a single run at the top of the program; the order between separate definitions in one chunk is the textual order they appear in this file.

⟨ ***Constants*** ⟩ ≡

```
const SOURCE_MAX      = 2097152;   # 2 MB: max .plm source size
const SOURCE_NAME_MAX = 1024;      # max source filename length
const OUT_MAX         = 4194304;   # 4 MB: MMIXAL output buffer
const STRING_POOL_MAX = 262144;    # 256 KB: decoded strings / normalised names
const TOKEN_MAX       = 131072;    # 128 Ki tokens
const NODE_POOL_MAX   = 8388608;   # 8 MB: AST node arena (octa-granular)
const INT_BUF_SIZE    = 32;        # scratch for integer-to-decimal conversion
```

Every token occupies a fixed 32-byte record. The five fields are packed into four octas:

Offset	Field	Meaning
0	TK_KIND	token kind (see the token-kinds section)
8	TK_VALUE	decoded integer value *or* string-pool offset
16	TK_LEN	string length (0 if the value is numeric)
24	TK_LINE	source line, 1-based (low 32 bits); column in high 32

Packing line and column into a single octa keeps the record on a multiple of 8 bytes, which matters because `TokenBuf` is indexed by multiplying a token index by TOKEN_ENTRY_SIZE.

⟨ ***Constants*** ⟩ ≡

```
const TOKEN_ENTRY_SIZE = 32;
const TOKEN_BUF_SIZE   = TOKEN_MAX * TOKEN_ENTRY_SIZE;
# Field offsets within a token:
const TK_KIND  = 0;
const TK_VALUE = 8;
const TK_LEN   = 16;
const TK_LINE  = 24;
```

An AST node has a variable-sized payload but a fixed header. The header is eight octas (64 bytes):

Offset	Field	Meaning
0	ND_KIND	AST node kind (assigned in the parser)
8	ND_LINE	source line of the leading token
16	ND_COL	source column of the leading token
24	ND_A	first payload slot (interpretation depends on kind)
32	ND_B	second payload slot
40	ND_C	third payload slot
48	ND_D	fourth payload slot
56	ND_NEXT	next-sibling offset, or 0 for the last in a list

Nodes that need more than four payload slots (for example procedure declarations with both parameter and body lists) reserve additional trailing octas by allocating a bigger entry; each parser routine that needs this documents how many extra octas it uses. An offset of 0 in ND_NEXT means "end of list", so node 0 is reserved and never allocated.

⟨ ***Constants*** ⟩ ≡

```
const NODE_HEADER_SIZE = 64;
const ND_KIND = 0;
const ND_LINE = 8;
const ND_COL  = 16;


const ND_A    = 24;
const ND_B    = 32;
const ND_C    = 40;
const ND_D    = 48;
const ND_NEXT = 56;

const ALIAS_ENTRY_SIZE = 24;
const ALIAS_NAME_OFF   = 0;
const ALIAS_NAME_LEN   = 8;
const ALIAS_REG_NUM    = 16;
const ALIAS_MAX        = 4096;
# Generator symbol table.  One entry per uniquely-named identifier
# encountered by the generator pre-passes; flag bits record what
# kinds of declaration apply to the name, |SYM_CONST_EXPR| holds the
# AST handle for `const' bindings, |SYM_DUP_COUNT| holds the
# cross-procedure occurrence count for proc-local labels.  See the
# "Generator: design overview" section for the full story.
const SYM_NAME_OFF   = 0;      # string-pool offset of the (normalized) name
const SYM_NAME_LEN   = 8;      # string-pool length of the (normalized) name
const SYM_FLAGS      = 16;     # bitmap of |SYM_F_*|
const SYM_CONST_EXPR = 24;     # AST handle, valid when |SYM_F_CONST| set
const SYM_DUP_COUNT  = 32;     # cross-procedure label occurrence count
const SYM_ENTRY_SIZE = 40;
const SYM_MAX        = 2048;

# Hash table over |$SymBuf|.  |$SymHashTbl| is a parallel array of
# |SYM_HASH_SIZE| slots; each slot holds either zero (empty) or the
# address of a |$SymBuf| entry.  |SYM_HASH_SIZE| is a power of two
# at least twice |SYM_MAX|, keeping linear-probing load below 0.5.
const SYM_HASH_SIZE = 4096;
const SYM_HASH_MASK = 4095;

const SYM_F_DATA       = 1;    # appears as a top-level byte/reserve/const label
const SYM_F_CONST      = 2;    # bound by `const x = expr'
const SYM_F_GREG       = 4;    # allocated via `greg'
const SYM_F_ADDR_TAKEN = 8;    # address taken anywhere in the program

# Per-procedure label scratch.  Reset at the start of each |def|;
# populated by |CollectProcLabels|; consulted by |ProcLabelMode|
# during emit to decide whether a label needs proc-name mangling.
const PROC_LABEL_ENTRY_SIZE = 16;
const PROC_LABEL_OFF        = 0;
const PROC_LABEL_LEN        = 8;
const PROC_LABEL_MAX        = 512;
```

AST node kinds. Grouped by syntactic role: program and segment roots first, then declarations, then statements, then expressions and leaf terminals. The codes are the stable interface between

parser and generator; add new kinds at the end of the appropriate block rather than renumbering existing ones. Code 0 is reserved so that a zeroed node header reliably signals "uninitialised".

⟨ ***Constants*** ⟩ ≡

```
# Program / segment structure.
const NK_PROGRAM   =  1;    # root; ND_A = first top-level decl
const NK_SEGDATA   =  2;    # segment data $Base;
const NK_SEGCODE   =  3;    # segment code;


# Top-level declarations.
const NK_GREG      = 10;    # greg $name[SIZE]; or greg $name = val;
const NK_CONST     = 11;    # const NAME = expr;
const NK_DEF       = 12;    # def Name(params) -> outs: body end
const NK_DATADIR   = 13;    # byte/wyde/tetra/octa Label items...
const NK_LOC       = 14;    # loc address;
const NK_RESERVE   = 15;    # reserve N;
const NK_PREFIX    = 16;    # prefix Prefix;
const NK_LOCAL     = 17;    # local NAME;
# Historical node-kind slot 18 is intentionally unreachable: `extern`
# has been removed, but later node-kind codes stay stable so the
# bootstrap parser/generator does not need a full renumbering pass.
const NK_POOL      = 19;    # pool;
# Statements.
const NK_REGDECL   = 30;    # register $a, $b, ...;
const NK_ASSIGN    = 32;    # $dst = expr;
const NK_STORE     = 33;    # [ptr+off].size = expr;
const NK_CALL      = 34;    # call Proc(args);
const NK_CALLASN   = 35;    # $dst = call Proc(args);
const NK_RETURN    = 36;    # return; / return $r;
const NK_GOTO      = 37;    # goto Label; / goto Label if cond;
const NK_IF        = 38;    # if cond: ... end (ND_A=cond, ND_B=body, ND_C=elif chain,
 ND_D=else)
const NK_ELIF      = 39;    # elif branch in chain
const NK_WHILE     = 40;    # while cond: ... end
const NK_LOOP      = 41;    # loop: ... end
const NK_DO        = 42;    # do: ... end
const NK_BREAK     = 43;    # break;
const NK_CONTINUE  = 44;    # continue;
const NK_LABELSTMT = 45;    # Label:
const NK_TRAP      = 46;    # trap N;
const NK_SAVE      = 47;    # save $r;
const NK_UNSAVE    = 48;    # unsave $r;
const NK_ROUNDING  = 49;    # rounding mode;
const NK_PRELD     = 50;    # preld/prego/prest/syncd/syncid/cswap/ldvts
const NK_BEGIN     = 51;    # begin: ... end (block)
const NK_SYNC      = 52;    # sync x;
const NK_TRIP      = 53;    # trip x, y, z;
const NK_RESUME    = 54;    # resume; / resume z;
# Expressions.
const NK_BINOP     = 60;    # ND_A=lhs, ND_B=rhs, ND_C=op-code
const NK_UNOP      = 61;    # ND_A=operand, ND_C=op-code
const NK_LOAD      = 62;    # [ptr+off].size load, ND_A=ptr expr, ND_B=off expr,
 ND_C=size-code
const NK_GETA      = 63;    # &Label, ND_A = label-pool-offset
const NK_PAREN     = 64;    # (expr) - parser convenience; may collapse to child
const NK_COND      = 65;    # "$x is <relop>" condition inside goto/if
const NK_WYDESET   = 66;    # $dst orh= expr (wyde-op assignment as statement)
# Terminals.
const NK_IMM       = 80;    # integer literal; ND_A = value
const NK_FLOAT     = 81;    # float literal; ND_A = pool offset of raw text
```

```
const NK_STRING    = 82;    # string literal; ND_A = pool offset, ND_B = len
const NK_REG       = 83;    # $N register; ND_A = register number
const NK_REGNAME   = 84;    # $name; ND_A = pool offset, ND_B = len
const NK_SREG      = 85;    # $rA..$rZZ; ND_A = MMIX sreg index
const NK_LABELREF  = 86;    # bare identifier used as label; ND_A = pool offset, ND_B = len
# Helper / list nodes (often trailing octas carry counts).
const NK_PARAMLIST = 90;    # ND_A = first param (linked by ND_NEXT)
const NK_ARGLIST   = 91;    # call argument list
const NK_PARAM     = 92;    # one formal parameter; ND_A = name-offset, ND_B = len
const NK_ITEMLIST  = 93;    # data-directive item list
```

3. Data segment.

All program state lives in the data segment. The order in which the buffers appear below follows the pipeline: input comes in through `SourceBuf`, then `TokenBuf` and `StringPool` accumulate lexer output, then `NodeBuf` accumulates parser output, and finally `OutputBuf` is assembled by the code generator.

Each buffer is paired with a counter that marks the first unused byte (or, for `TokenBuf` and `NodeBuf`, the first unused record). The counters start at 0 and only ever grow during a single compilation run.

Like `@<Constants@>=`, the `@<Data segment@>=` chunk is opened here and extended in many later sections. Every later occurrence adds fresh state buffers introduced by that section (lexer tables, error message strings, generator opcode tables, etc.); the tangled output places all of them inside a single `segment data $DataBase;` block in textual order.

⟨ ***Data segment*** ⟩ ≡

```
segment  data $DataBase;
# MMIX FREAD/FWRITE argument blocks (buffer pointer, byte count).
greg $FReadArgs[16];
greg $FWriteArgs[16];


# Raw source input and its captured filename.
greg $SourceBuf[SOURCE_MAX];
greg $SourceNameBuf[SOURCE_NAME_MAX];
# Lexer output.
greg $TokenBuf[TOKEN_BUF_SIZE];
greg $StringPool[STRING_POOL_MAX];
# Parser output.
greg $NodeBuf[NODE_POOL_MAX];
# Generator output and scratch for integer-to-decimal conversion.
greg $OutputBuf[OUT_MAX];
greg $IntBuf[INT_BUF_SIZE];
# Counters (all start at 0).
greg $src_len      = 0;     # bytes currently in SourceBuf
greg $tok_cnt      = 0;     # tokens currently in TokenBuf
greg $str_end      = 0;     # first unused byte in StringPool
greg $node_end     = 0;     # first unused byte in NodeBuf (starts at NODE_HEADER_SIZE to
  reserve node 0)
greg $out_end      = 0;     # first unused byte in OutputBuf
# Register-usage bitmaps.  Declared here, ahead of any |byte|
# directive, so they inherit octa alignment from the surrounding
# octa-sized |greg| arrays.  |MaxRegUsed| relies on this to scan
# eight bytes at a time via an octa load.
greg $GlobalRegUsed[256];
```

```
greg $LocalRegUsed[256];
# Default source name used when argv[1] is absent.
byte DefaultSourceName "stdin.plm", 0;
# Error messages (null-terminated).
byte ErrSrcFull   "plmmix-compiler: source reaches input-buffer limit\n", 0;
byte ErrOutFull   "plmmix-compiler: output buffer full\n", 0;
byte ErrStrFull   "plmmix-compiler: string pool full\n", 0;
byte ErrAtLine    "line ", 0;
byte ErrAtCol     ", col ", 0;
byte ErrAtSep     ": ", 0;
```

4. Main program.

The top-level flow has one step per pipeline phase described in the introduction, plus a final write-out of `OutputBuf`. Each phase operates on the shared arenas declared in the data-segment section; no data is passed explicitly between them.

⟨ ***Program*** ⟩ ≡

```
⟨ Constants ⟩
⟨ Data segment ⟩
 segment  code;
 register $ioReg = $255;
 call InitSourceName($0, $1);
 call ReadAll();
 # Reserve node 0 so that a zero ND_NEXT means end-of-list.
$node_end = NODE_HEADER_SIZE;
 call Tokenize();
 call ParseProgram();
 call Generate();
⟨ Write output to stdout ⟩
 trap 0, HALT, 0;
⟨ Procedures ⟩
```

After the generator finishes, `$out_end` bytes are waiting in `OutputBuf`. One `FWRITE` call flushes them all to standard output.

⟨ ***Write output to stdout*** ⟩ ≡

```
$ioReg = $OutputBuf;
[$FWriteArgs].o       = $ioReg;
$ioReg = $out_end;
[$FWriteArgs + 8].o   = $ioReg;
$ioReg = $FWriteArgs;
 trap 0, FWRITE, STDOUT;
```

The procedure-list chunk is collected here because it is the structural outline for the remainder of the file. Each phase of the pipeline appends its procedures to this list, in the order that they will appear in the tangled output. The early entries (`ReadAll`, `InitSourceName`, the string and output utilities, and the error handler) are all defined in the present file; the lexer, parser, and generator procedures are defined in the later sections of this file.

⟨ ***Procedures*** ⟩ ≡

```
⟨ ReadAll procedure ⟩
⟨ InitSourceName procedure ⟩
⟨ String utilities ⟩
⟨ Integer conversion ⟩
⟨ Output helpers ⟩
⟨ Error handler ⟩
⟨ Lexer procedures ⟩
⟨ Parser procedures ⟩
⟨ Generator procedures ⟩
```

5. Reading the source.

`ReadAll` fills `SourceBuf` from standard input in one `FREAD` call, then stores the actual byte count in `$src_len`.

MMIX's `FREAD` trap takes a two-octa argument block (buffer pointer followed by requested byte count) and returns *requested_count* − *bytes_transferred* in register `$255`, so zero on a complete read. We recover the actual count as `SOURCE_MAX + return_value`. If the source exactly fills `SOURCE_MAX` bytes we cannot tell whether more data was truncated, so we treat that case as an overflow error rather than silently producing wrong output.

⟨ ***ReadAll procedure*** ⟩ ≡

```
def ReadAll():
  register $tmp;
 [$FReadArgs].o = $SourceBuf;
 $tmp= SOURCE_MAX;
 [$FReadArgs + 8].o = $tmp;
 $ioReg = $FReadArgs;
  trap 0, FREAD, STDIN;
 $tmp = SOURCE_MAX;
 $src_len = $ioReg + $tmp;
  goto RASizeOk  if $src_len  is  nonneg;
  call ErrHalt(&ErrSrcFull);
RASizeOk:
 $tmp = SOURCE_MAX;
  if $src_len >=u $tmp:
    # Exactly SOURCE_MAX bytes were read; there may be more we couldn't fit.
    call ErrHalt(&ErrSrcFull);
  end
  return;
end
```

`InitSourceName` copies `argv[1]` (if present and non-null) into `SourceNameBuf` for use in error messages; otherwise it uses the default name `stdin.plm`. The copy is bounded to `SOURCE_NAME_MAX - 1` bytes, leaving room for the null terminator. This procedure is a direct analogue of the one in `plmmix-tangle.plw`.

⟨ ***InitSourceName procedure*** ⟩ ≡

```
def InitSourceName($argc, $argv):
  register $src, $dst, $limit, $ch, $tmp;
 $dst = $SourceNameBuf;
 $limit = SOURCE_NAME_MAX;
 $limit = $limit - 1;
 $src = &DefaultSourceName;
  goto ISUseDefault  if $argc < 2;
 $tmp = $argv + 8;
 $src = [$tmp].o;
  goto ISUseDefault  if $src  is  zero;
  goto ISCopy;
ISUseDefault:
 $src = &DefaultSourceName;
ISCopy:
ISLoop:
  goto ISDone  if $limit  is  zero;
 $ch = [$src].bu;
  goto ISDone  if $ch  is  zero;
 [$dst].bu = $ch;
 $dst = $dst + 1;
 $src = $src + 1;
 $limit = $limit - 1;
  goto ISLoop;
ISDone:
 [$dst].bu = 0;
  return;
end
```

6. String utilities.

Four small subroutines form the complete string library shared by all phases of the compiler. `StrLen` counts bytes up to a null terminator; `StrEq` compares two byte strings given by pointer and length; `MemCpy` copies a byte range; and `PoolIntern` appends a byte range to `StringPool`, returning the offset at which it was stored.

String-pool offsets (rather than absolute pointers) are used in token and AST records so that the layout is position-independent and can be dumped for debugging without rewriting pointers.

`StrLen` counts bytes up to (but not including) the first null byte.

⟨ ***String utilities*** ⟩ ≡

```
def StrLen($s) -> $n:
  register $c;
 $n = 0;
SLLoop:
 $c = [$s].bu;
  if $c  is  zero:
    return $n;
  end
 $s = $s + 1;
 $n = $n + 1;
  goto SLLoop;
end
```

`StrEq` returns 1 if two byte strings of given lengths are equal, 0 otherwise. The fast path checks lengths first; only if they match are the bytes compared one at a time.

⟨ ***String utilities*** ⟩ ≡

```
def StrEq($p1, $l1, $p2, $l2) -> $r:
  register $c1, $c2, $tmp;
 $r = 0;
 $tmp = $l1 - $l2;
  if $tmp  is  nonzero:
    return $r;
  end
 SELoop:
  goto SEMatch  if $l1  is  zero;
 $c1 = [$p1].bu;
 $c2 = [$p2].bu;
 $tmp = $c1 - $c2;
  if $tmp  is  nonzero:
    return $r;
  end
 $p1 = $p1 + 1;
 $p2 = $p2 + 1;
 $l1 = $l1 - 1;
  goto SELoop;
 SEMatch:
 $r = 1;
  return $r;
end
```

`MemCpy` copies `$len` bytes from `$src` to `$dst`. The two regions must not overlap; the lexer and parser only ever copy from immutable `SourceBuf` into the (growing) `StringPool`, so overlap cannot arise.

⟨ ***String utilities*** ⟩ ≡

```
def MemCpy($dst, $src, $len):
  register $ch;
 MCLoop:
  if $len  is  zero:
    return;
  end
 $ch = [$src].bu;
 [$dst].bu = $ch;
 $src = $src + 1;
 $dst = $dst + 1;
 $len = $len - 1;
  goto MCLoop;
end
```

`PoolIntern` appends `$len` bytes starting at `$src` to `StringPool` and returns the offset (relative to `StringPool`) at which they were placed. The string is *not* null-terminated in the pool; tokens record the explicit length, so terminators would only waste space.

The same byte range may be interned multiple times; no deduplication is performed because the pool is large relative to typical token counts and because most tokens are numeric and never intern at all.

⟨ ***String utilities*** ⟩ ≡

```
def PoolIntern($src, $len) -> $offset:
  register $dst, $tmp;
 $tmp = $str_end + $len;
 $tmp = $tmp  cmpu STRING_POOL_MAX;
  if $tmp  is  pos:
    call ErrHalt(&ErrStrFull);
  end
 $offset = $str_end;
 $dst = $StringPool + $str_end;
  call MemCpy($dst, $src, $len);
 $str_end = $str_end + $len;
  return $offset;
end
```

`PoolAppendByte` appends a single byte to `StringPool` and advances `$str_end` by one. It is used by the string-literal scanner, which decodes escapes on the fly and therefore needs to grow a pool entry one byte at a time rather than copying a contiguous source slice. A pool-overflow check is performed on every append; callers do not need to reserve space in advance.

⟨ ***String utilities*** ⟩ ≡

```
def PoolAppendByte($b):
  register $tmp, $dst;
 $tmp = $str_end  cmpu STRING_POOL_MAX;
  if $tmp  is  nonneg:
    call ErrHalt(&ErrStrFull);
  end
 $dst = $StringPool + $str_end;
 [$dst].b = $b;
 $str_end = $str_end + 1;
  return;
end
```

7. Integer conversion.

`AppendInt` writes a non-negative integer to `OutputBuf` as decimal ASCII. The usual mod-10 loop would emit digits in reverse order; we stash them into `IntBuf` and then replay the buffer. The code handles `$n == 0` specially because the main loop would otherwise emit no digits at all.

A signed variant is not provided: every integer written by the generator (line numbers, register numbers, immediate operands) is either non-negative or already formatted with a leading minus sign by the caller.

⟨ ***Integer conversion*** ⟩ ≡

```
def AppendInt($n):
  register $count, $digit, $tmp, $ptr;
  if $n is zero:
    call AppendByte('0');

    return;
  end
 $count = 0;
 AILoop:
  goto AIEmit if $n is zero;
 $tmp = $n / 10;
 $digit = $tmp * 10;
 $digit = $n - $digit;
 $digit = $digit + '0';
 $ptr = $IntBuf + $count;
 [$ptr].bu = $digit;
 $count = $count + 1;
 $n = $tmp;
  goto AILoop;
 AIEmit:
  if $count is zero:
    return;
  end
 $count = $count - 1;
 $ptr = $IntBuf + $count;
 $digit = [$ptr].bu;
  call AppendByte($digit);
  goto AIEmit;
end
```

`AppendHexNibble` appends one uppercase hexadecimal digit for a value in the range 0..15. `AppendHexByte` writes a full MMIXAL hex byte literal in the form `#XX`.

⟨ ***Integer conversion*** ⟩ ≡

```
def AppendHexNibble($n):
  register $tmp;
 $tmp = $n - 10;
  goto AHNHex if $tmp is nonneg;
 $n = $n + '0';
  call AppendByte($n);
  return;
 AHNHex:
 $tmp = $tmp + 'A';
  call AppendByte($tmp);
  return;
end

def AppendHexByte($v):
  register $hi, $lo;
  call AppendByte(' #');
 $hi = $v >>u 4;
 $lo = $v & 15;
  call AppendHexNibble($hi);
  call AppendHexNibble($lo);
  return;
end
```

`AppendHexWyde` writes a four-digit hexadecimal immediate in MMIXAL form `#XXXX`, taking only the low 16 bits of `$v`.

⟨ ***Integer conversion*** ⟩ ≡

```
def AppendHexWyde($v):
  register $n;
  call AppendByte(' #');
 $n = $v >>u 12;
 $n = $n & 15;
  call AppendHexNibble($n);
 $n = $v >>u 8;
 $n = $n & 15;
  call AppendHexNibble($n);
 $n = $v >>u 4;
 $n = $n & 15;
  call AppendHexNibble($n);
 $n = $v & 15;
  call AppendHexNibble($n);
  return;
end
```

`AppendHexOcta` writes a full 64-bit hexadecimal literal as `#XXXXXXXXXXXXXXXX` (16 uppercase hex digits). This path is used for negative immediate values, whose internal two's-complement bit pattern must be preserved verbatim without signed decimal conversion.

⟨ ***Integer conversion*** ⟩ ≡

```
def AppendHexOcta($v):
  register $i, $shift, $nib;
  call AppendByte(' #');
 $i = 16;
 AHO_Loop:
  if $i  is  zero:
    return;
  end
 $i = $i - 1;
 $shift = $i << 2;
 $nib = $v >>u $shift;
 $nib = $nib & 15;
  call AppendHexNibble($nib);
  goto AHO_Loop;
end
```

8. Output helpers.

All MMIXAL text that the generator emits flows through three procedures. `AppendByte` is the hot path, called for every single character of the output; it is kept short for that reason. `AppendBytes` copies an explicit range; `AppendCStr` copies a null-terminated string by computing its length with `StrLen`.

Every overflow check halts the program through `ErrHalt`. The checks are duplicated across `AppendByte` and `AppendBytes` rather than funnelled through a single gate because both proce-

dures are on the generator's fast path and the branch predictor handles a forward-not-taken check better than a function call.

⟨ ***Output helpers*** ⟩ ≡

```
def AppendByte($ch):
  register $tmp;
 $tmp = $out_end  cmpu OUT_MAX;
  if $tmp  is  nonneg:


    call ErrHalt(&ErrOutFull);
  end
 $OutputBuf[$out_end].bu = $ch;
 $out_end = $out_end + 1;
  return;
end

def AppendBytes($src, $len):
  register $ch, $tmp;
 $tmp = $out_end + $len;
 $tmp = $tmp  cmpu OUT_MAX;
  if $tmp  is  pos:
    call ErrHalt(&ErrOutFull);
  end
 ABLoop:
  if $len  is  zero:
    return;
  end
 $ch = [$src].bu;
 $OutputBuf[$out_end].bu = $ch;
 $out_end = $out_end + 1;
 $src = $src + 1;
 $len = $len - 1;
  goto ABLoop;
end

def AppendCStr($ptr):
  register $len;
 $len =  call StrLen($ptr);
  call AppendBytes($ptr, $len);
  return;
end
```

9. Error handling.

Every error in this program is fatal: once an anomaly is detected we write a diagnostic to standard error through `FPUTS` and exit with status 1. Because the four buffers are always left in a partially consistent state on exit, the caller must take care not to trust any output that was written before a crash; that is why `OutputBuf` is flushed only in the success path of `main`, after `trap 0, HALT, 0` would otherwise execute.

Each call site loads the address of a null-terminated message (defined in the data-segment section) into `$tmp` and passes it as the single argument to `ErrHalt`. Phase-specific messages that need variable content (line numbers, offending tokens) are printed directly by their owning phase through `AppendByte`-style helpers before calling a dedicated fatal routine defined alongside them.

⟨ ***Error handler*** ⟩ ≡

```
def ErrHalt($msg):
 $ioReg = $msg;
  trap 0, FPUTS, STDERR;
  trap 0, HALT, 1;
end
```

⟨ ***Error handler*** ⟩ ≡

```
def ErrHaltAt($msg, $line, $col):
 $out_end = 0;
  call AppendCStr(&ErrAtLine);
  call AppendInt($line);
  call AppendCStr(&ErrAtCol);
  call AppendInt($col);
  call AppendCStr(&ErrAtSep);
  call AppendCStr($msg);
 $ioReg = $OutputBuf;
 [$FWriteArgs].o = $ioReg;
 $ioReg = $out_end;
 [$FWriteArgs + 8].o = $ioReg;
 $ioReg = $FWriteArgs;
  trap 0, FWRITE, STDERR;
  trap 0, HALT, 1;
end
```

10. Token kinds.

Every token emitted by the lexer carries a numeric `kind` that is tested by the parser. The codes are grouped by role: punctuation first, then compound assignment and comparison operators, then arithmetic, then terminals (registers, literals, identifiers). A single `TK_KW` code is reused for every reserved keyword; the parser distinguishes keywords by their interned-string payload.

The symbolic names below are kept short and regular so the parser, generator, and diagnostics can refer to token kinds consistently.

⟨ ***Constants*** ⟩ +≡

```
# Token kinds.
const TK_EOF      =  0;
# Punctuation.
const TK_LPAREN   =  1;     # (


const TK_RPAREN   =  2;     # )
const TK_LBRACKET =  3;     # [
const TK_RBRACKET =  4;     # ]
const TK_COMMA    =  5;     # ,
const TK_SEMI     =  6;     # ;
const TK_COLON    =  7;     # :
const TK_AMP      =  8;     # & (dual-use: geta prefix and bitwise AND)
# Assignment and equality.
const TK_EQUALS   = 10;     # =
const TK_ZASSIGN  = 11;     # ?=
const TK_ARROW    = 12;     # ->
# Integer comparison.
```

```
const TK_EQ       = 13;     # ==
const TK_NEQ      = 14;     # !=
const TK_LT       = 15;     # <
const TK_LEQ      = 16;     # <=
const TK_GT       = 17;     # >
const TK_GEQ      = 18;     # >=
const TK_LTU      = 19;     # <u
const TK_LEQU     = 20;     # <=u
const TK_GTU      = 21;     # >u
const TK_GEQU     = 22;     # >=u
# Floating-point comparison.
const TK_EQF      = 23;     # ==.
const TK_NEQF     = 24;     # !=.
const TK_LTF      = 25;     # <.
const TK_LEQF     = 26;     # <=.
const TK_GTF      = 27;     # >.
const TK_GEQF     = 28;     # >=.
# Integer arithmetic and bitwise.
const TK_PLUS     = 30;     # +
const TK_MINUS    = 31;     # -
const TK_STAR     = 32;     # *
const TK_SLASH    = 33;     # /
const TK_PERCENT  = 34;     # %
const TK_PLUSU    = 35;     # +u
const TK_MINUSU   = 36;     # -u
const TK_STARU    = 37;     # *u
const TK_SLASHU   = 38;     # /u
# Floating-point arithmetic.
const TK_FADD     = 39;     # +.
const TK_FSUB     = 40;     # -.
const TK_FMUL     = 41;     # *.
const TK_FDIV     = 42;     # /.
const TK_FREM     = 43;     # %.
# Bitwise and shift.
const TK_PIPE     = 44;     # |
const TK_CARET    = 45;     # ^
const TK_SHL      = 46;     # <<
const TK_SHR      = 47;     # >>
const TK_SHLU     = 48;     # <<u
const TK_SHRU     = 49;     # >>u
# Scaled unsigned add (digit-prefixed).
const TK_2ADDU    = 50;
const TK_4ADDU    = 51;
const TK_8ADDU    = 52;
const TK_16ADDU   = 53;
# Terminals.
const TK_REGISTER = 60;     # $N          ; TK_VALUE = N
const TK_REGNAME  = 61;     # $name       ; TK_VALUE = pool offset, TK_LEN = len
const TK_SREG     = 62;     # $rA etc.    ; TK_VALUE = MMIX sreg index
const TK_IMMEDIATE= 63;     # integer     ; TK_VALUE = decoded integer
const TK_FLOAT    = 64;     # float       ; TK_VALUE = pool offset of raw text, TK_LEN = len
const TK_STRING   = 65;     # "..."       ; TK_VALUE = pool offset, TK_LEN = len
const TK_LABEL    = 66;     # identifier  ; TK_VALUE = pool offset, TK_LEN = len
const TK_KW       = 67;     # keyword     ; TK_VALUE = keyword code
const TK_WYDEOP   = 68;     # orh= etc.   ; TK_VALUE = wyde-op code
const TK_UNARYOP  = 69;     # sqrt. etc.  ; TK_VALUE = unary-op code
const TK_SIZE     = 70;     # .b .bu etc. ; TK_VALUE = size-suffix code
```

Keyword codes. When the lexer recognises a reserved identifier (`def`, `end`, `if`, `while`, ...), it emits a `TK_KW` token whose `TK_VALUE` field is the keyword's index in the master table. The indices

below are one-based and must match the order of entries in `KeywordTable` exactly: the lookup returns the running position, so reordering the table without updating these constants would silently mis-classify every keyword. Index 0 is reserved for "not a keyword", which `KeywordLookup` returns for ordinary identifiers.

The parser uses these constants to dispatch on keyword identity without a second string comparison. Named binary operators (`andn`, `cmp`, `sadd`, ...) are keywords too: the parser promotes them to binary-operator AST nodes in expression context.

⟨ ***Constants*** ⟩ +≡

```
# Keyword codes (index into KeywordTable, 1-based; 0 = not a keyword).
const KW_DEF      =  1;
const KW_END      =  2;
const KW_BEGIN    =  3;


const KW_IF       =  4;
const KW_ELIF     =  5;
const KW_ELSE     =  6;
const KW_WHILE    =  7;
const KW_DO       =  8;
const KW_LOOP     =  9;
const KW_BREAK    = 10;
const KW_CONTINUE = 11;
const KW_GOTO     = 12;
const KW_CALL     = 13;
const KW_RETURN   = 14;
const KW_GREG     = 15;
const KW_DATA     = 16;
const KW_LOC      = 17;
const KW_RESERVE  = 18;
const KW_REGISTER = 19;
# Historical keyword slot 20 is intentionally unreachable: `scratch`
# is no longer reserved, but later keyword codes stay stable so the
# bootstrap parser/generator does not need a full renumbering pass.
const KW_CONST    = 21;
const KW_PREFIX   = 22;
const KW_LOCAL    = 23;
# Historical keyword slot 24 is intentionally unreachable: `extern`
# is no longer reserved, but later keyword codes stay stable so the
# bootstrap parser/generator does not need a full renumbering pass.
const KW_SEGMENT  = 25;
const KW_CODE     = 26;
const KW_POOL     = 27;
const KW_BYTE     = 28;
const KW_WYDE     = 29;
const KW_TETRA    = 30;
const KW_OCTA     = 31;
const KW_ROUNDING = 32;
const KW_SAVE     = 33;
const KW_UNSAVE   = 34;
const KW_PRELD    = 35;
const KW_PREGO    = 36;
const KW_PREST    = 37;
const KW_SYNCD    = 38;
const KW_SYNCID   = 39;
const KW_SYNC     = 40;
const KW_CSWAP    = 41;
const KW_LDVTS    = 42;
const KW_TRAP     = 43;
```

```
const KW_TRIP     = 44;
const KW_RESUME   = 45;
const KW_LIKELY   = 46;
const KW_UNLIKELY = 47;
const KW_IS       = 48;
const KW_NEG      = 49;
const KW_ZERO     = 50;
const KW_POS      = 51;
const KW_ODD      = 52;
const KW_NONNEG   = 53;
const KW_NONZERO  = 54;
const KW_NONPOS   = 55;
const KW_EVEN     = 56;
const KW_NEAREST  = 57;
const KW_UP       = 58;
const KW_DOWN     = 59;
const KW_ANDN     = 60;
const KW_ORN      = 61;
const KW_NAND     = 62;
const KW_NOR      = 63;
const KW_NXOR     = 64;
const KW_MOR      = 65;
const KW_MXOR     = 66;
const KW_MUX      = 67;
const KW_SADD     = 68;
const KW_BDIF     = 69;
const KW_WDIF     = 70;
const KW_TDIF     = 71;
const KW_ODIF     = 72;
const KW_CMP      = 73;
const KW_CMPU     = 74;
# Dotted named-comparison keywords (recognised via ident+'.' post-check).
const KW_CMPEDOT  = 75;
const KW_EQLEDOT  = 76;
const KW_UNEDOT   = 77;
```

Unary-operator, wyde-op and size-suffix codes. These three parallel code spaces correspond to the `TK_UNARYOP`, `TK_WYDEOP` and `TK_SIZE` token kinds. Each set is 1-based and dense; the lookup tables in the data segment are ordered to match these constants exactly, so `TableLookup`'s 1-based index return can be stored directly in `TK_VALUE`.

⟨ ***Constants*** ⟩ +≡

```
# Unary-operator codes (TK_UNARYOP TK_VALUE).
const UNARY_SQRT   = 1;
const UNARY_INT    = 2;
const UNARY_FLOT   = 3;


const UNARY_FLOTU  = 4;
const UNARY_FIX    = 5;
const UNARY_FIXU   = 6;
const UNARY_SFLOT  = 7;
const UNARY_SFLOTU = 8;
# Wyde-operator codes (TK_WYDEOP TK_VALUE).
const WYDE_ANDNH   = 1;
const WYDE_ANDNL   = 2;
const WYDE_ANDNMH  = 3;
const WYDE_ANDNML  = 4;
const WYDE_ORH     = 5;
```

```
const WYDE_ORL     = 6;
const WYDE_ORMH    = 7;
const WYDE_ORML    = 8;
# Size-suffix codes (TK_SIZE TK_VALUE).
const SIZE_B       =  1;
const SIZE_BU      =  2;
const SIZE_W       =  3;
const SIZE_WU      =  4;
const SIZE_T       =  5;
const SIZE_TU      =  6;
const SIZE_O       =  7;
const SIZE_SF      =  8;
const SIZE_HT      =  9;
const SIZE_UNC     = 10;
```

The lexer needs three additional error messages in the data segment beyond the generic buffer-full ones declared in the data-segment section. They are collected here because they are produced only by the lexer; the parser and generator use their own.

⟨ ***Data segment*** ⟩ +≡

```
byte ErrLexChar  "plmmix-compiler: unexpected character in source\n", 0;
byte ErrLexUnterm "plmmix-compiler: unterminated literal\n", 0;
byte ErrLexEscape "plmmix-compiler: unknown escape sequence in literal\n", 0;
```

The keyword table. Each entry is a length byte followed by that many ASCII bytes; a zero length byte terminates the table. Entries appear in the same order as the `KW_xxx` constants defined in the token-kinds section: `KeywordLookup` walks the table linearly and returns the 1-based position of a matching entry, or 0 for a non-keyword identifier.

A length-prefixed layout is used (rather than zero-terminated strings) because the ident scanner already knows the lexeme length from the `$scan`–`$tok_start` subtraction, so length comparison is a single octa subtract before any character comparison. The longest keyword, `continue`, is eight bytes, so one byte of length is more than enough.

The entries are grouped on successive `byte` lines for readability; all lines append to the same running data position because only the first line carries the `KeywordTable` label.

Keyword slots 20 and 24 are intentionally filled with non-identifier sentinels (`scra!ch` and `exte!n`) so that source text `scratch` and `extern` no longer tokenise as reserved words, while the historical 1-based codes for later keywords remain unchanged.

⟨ ***Data segment*** ⟩ +≡

```
byte KeywordTable 3,"def", 3,"end", 5,"begin", 2,"if", 4,"elif", 4,"else";
byte KeywordTable1 5,"while", 2,"do", 4,"loop", 5,"break", 8,"continue";
byte KeywordTable2 4,"goto", 4,"call", 6,"return";
byte KeywordTable3 4,"greg", 4,"data", 3,"loc", 7,"reserve", 8,"register";
byte KeywordTable4 7,"scra!ch", 5,"const", 6,"prefix", 5,"local", 6,"exte!n";
byte KeywordTable5 7,"segment", 4,"code", 4,"pool", 4,"byte", 4,"wyde", 5,"tetra", 4,"octa";
byte KeywordTable6 8,"rounding", 4,"save", 6,"unsave";
byte KeywordTable7 5,"preld", 5,"prego", 5,"prest", 5,"syncd", 6,"syncid", 4,"sync";
byte KeywordTable8 5,"cswap", 5,"ldvts", 4,"trap", 4,"trip", 6,"resume";
byte KeywordTable9 6,"likely", 8,"unlikely";
byte KeywordTable10 2,"is", 3,"neg", 4,"zero", 3,"pos", 3,"odd";
byte KeywordTable11 6,"nonneg", 7,"nonzero", 6,"nonpos", 4,"even";
byte KeywordTable12 7,"nearest", 2,"up", 4,"down";
```

```
byte KeywordTable13 4,"andn", 3,"orn", 4,"nand", 3,"nor", 4,"nxor";
byte KeywordTable14 3,"mor", 4,"mxor", 3,"mux", 4,"sadd";
byte KeywordTable15 4,"bdif", 4,"wdif", 4,"tdif", 4,"odif";
byte KeywordTable16 3,"cmp", 4,"cmpu";
byte KeywordTable17 0;
```

The special-register table. Each entry is a one-byte length, the ASCII name (without the `r` prefix and without the leading `$`), then a one-byte MMIX special-register index (0–31). A zero length byte terminates the table. `SregLookup` returns `index + 1` on a hit (so that zero can unambiguously signal miss) and 0 otherwise.

The table is searched only when the dollar-prefixed lexeme begins with the letter `r`; an empty rest (the bare `$r`) returns miss, and the lexer then treats the token as an ordinary `TK_REGNAME`. Entries are grouped to mirror the MMIX documentation order, but `SregLookup` compares by length first so the grouping affects only readability, not correctness.

⟨ ***Data segment*** ⟩ +≡

```
byte SregTable 1,"B",0, 1,"D",1, 1,"E",2, 1,"H",3, 1,"J",4, 1,"M",5, 1,"R",6;
byte SregTable1 2,"BB",7, 1,"C",8, 1,"N",9, 1,"O",10, 1,"S",11, 1,"I",12;
byte SregTable2 1,"T",13, 2,"TT",14, 1,"K",15, 1,"Q",16, 1,"U",17, 1,"V",18;
byte SregTable3 1,"G",19, 1,"L",20, 1,"A",21, 1,"F",22, 1,"P",23;
byte SregTable4 1,"W",24, 1,"X",25, 1,"Y",26, 1,"Z",27;
byte SregTable5 2,"WW",28, 2,"XX",29, 2,"YY",30, 2,"ZZ",31;
byte SregTable6 0;
```

Suffix-aware lexer tables. Four small tables drive the post-identifier classification phase and the dot-prefixed scanner:

`UnaryOpTable` maps the seven ident-like unary float operators (`sqrt`, `int`, `flot`, `flotu`, `fix`, `fixu`, `sflot`, `sflotu`) to `TK_UNARYOP` codes 1–8. The lookup is performed only when an identifier is followed immediately by `..`

`DotKwTable` maps the three named-comparison keywords `cmpe`, `eqle`, `une` (all followed by `.`) to their `KW_*` constants via an embedded index byte, using the same layout as `SregTable`.

`WydeOpTable` maps the eight wyde-immediate assignment operators (`andnh=`, ..., `orml=`) to `TK_WYDEOP` codes 1–8. The lookup is triggered when an identifier is followed by `=`.

`SizeTable` maps the ten dotted size suffixes `.b`, `.bu`, ..., `.unc` to `TK_SIZE` codes 1–10. It is the only dot-prefixed lookup; `.`-followed-by-anything-else is an error.

All four tables use the length-prefixed entry format shared with `KeywordTable` and terminate with a zero length byte.

⟨ ***Data segment*** ⟩ +≡

```
byte UnaryOpTable 4,"sqrt", 3,"int", 4,"flot", 5,"flotu";
byte UnaryOpTable1 3,"fix", 4,"fixu", 5,"sflot", 6,"sflotu";
byte UnaryOpTable2 0;
```

⟨***Data segment***⟩ +≡

```
byte DotKwTable 4,"cmpe",75, 4,"eqle",76, 3,"une",77;
byte DotKwTable1 0;
```

⟨***Data segment***⟩ +≡

```
byte WydeOpTable 5,"andnh", 5,"andnl", 6,"andnmh", 6,"andnml";
byte WydeOpTable1 3,"orh", 3,"orl", 4,"ormh", 4,"orml";
byte WydeOpTable2 0;
```

⟨***Data segment***⟩ +≡

```
byte SizeTable 1,"b", 2,"bu", 1,"w", 2,"wu", 1,"t", 2,"tu";
byte SizeTable1 1,"o", 2,"sf", 2,"ht", 3,"unc";
byte SizeTable2 0;
```

11. Lexer: overview and main loop.

`Tokenize` makes a single left-to-right pass over `SourceBuf`, appending one fixed-size record to `TokenBuf` per recognised token and interning any string-valued payload (identifier names, raw string bodies) into `StringPool`. A final `TK_EOF` terminator is always emitted so that the parser can look one token ahead without a bounds check.

The pass is a three-phase inner loop. First, whitespace and line comments are skipped; this is the only place where `$line` and `$line_start` are updated. Second, the byte at `$scan` is classified by dispatch on its value: single-character punctuation emits immediately, compound operators consume one or two more bytes, and complex literals (numbers, strings, identifiers, register sigils) hand off to specialised scanners. Third, control returns to the top of the loop until the input is exhausted.

A line comment begins with `#` at the start of a lexeme, but the same character also begins a hex-literal (`#FF`, `#2000000000000000`). The two are distinguished by the byte that follows: a hex digit means the `#` is part of a literal, anything else means a comment running to the end of line. This check is done once in |@<Tokenize: skip whitespace and comments@>|.

The key state variables are:

`$scan`, `$send` Read cursor and one-past-end of `SourceBuf`.

`$line`, `$line_start` Current source line (1-based) and pointer to the first byte of that line. A token's column is computed as `$scan - $line_start + 1` at the moment the token begins.

`$tok_line`, `$tok_col` Line and column captured at the start of the current token; passed to `EmitToken` so every token carries the position of its first byte, not its last.

`$ch`, `$next` Scratch registers for the current byte and (lookahead) next byte.

`$tmp` General scratch, mostly used for subtract-then-branch-on-zero comparisons.

⟨ ***Lexer procedures*** ⟩ ≡

```
def Tokenize():
  register $scan, $send, $ch, $next, $line, $line_start;
  register $tok_start, $tok_line, $tok_col, $tmp;
  register $idlen, $idvalue, $idkind, $numval, $name_start;
 ⟨Tokenize: initialize scan state⟩
 ⟨Tokenize: main loop⟩
 ⟨Tokenize: skip whitespace and comments⟩
 ⟨Tokenize: capture token position⟩
 ⟨Tokenize: punctuation dispatch⟩
 ⟨Tokenize: operator dispatch⟩
 ⟨Tokenize: deferred scanners⟩
 TLDone:
 ⟨Tokenize: emit EOF⟩
end
```

Initialisation sets `$scan` to the first byte of `SourceBuf` and `$send` to one past the last valid byte. The line counter starts at 1, and `$line_start` points at the same byte as `$scan`, so the very first token on line 1 reports column 1.

⟨ ***Tokenize: initialize scan state*** ⟩ ≡

```
$scan       = $SourceBuf;
$send       = $SourceBuf + $src_len;
$line       = 1;
$line_start = $SourceBuf;
$tok_line   = 1;
$tok_col    = 1;
```

The outer loop simply alternates between the whitespace-skip phase and the token-dispatch phase until the input is exhausted. The labels `TLMain`, `TLSkip`, and `TLDispatch` are all entered by `goto`; there is no fall-through between them, which makes the structure of the scanner easy to follow despite its size.

⟨ ***Tokenize: main loop*** ⟩ ≡

```
TLMain:
 goto TLSkip;
```

Whitespace is space, tab, carriage return, and newline. Only newline advances `$line`, and it also resets `$line_start` so that subsequent column computations refer to the new line. Line comments (`#` not followed by a hex digit) are consumed inline, leaving the terminating newline in place so that the outer case handles the line bump uniformly.

⟨ ***Tokenize: skip whitespace and comments*** ⟩ ≡

```
TLSkip:
 goto TLDone  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - ' ';


 goto TLSkipWS  if $tmp  is  zero;
$tmp = $ch - 9;            # tab
 goto TLSkipWS  if $tmp  is  zero;
$tmp = $ch - 13;           # carriage return
 goto TLSkipWS  if $tmp  is  zero;
$tmp = $ch - 10;           # newline
 goto TLSkipNL  if $tmp  is  zero;
$tmp = $ch - ' #';
 goto TLMaybeComment  if $tmp  is  zero;
 goto TLDispatch;
TLSkipWS:
$scan = $scan + 1;
 goto TLSkip;
TLSkipNL:
$scan = $scan + 1;
$line = $line + 1;
$line_start = $scan;
 goto TLSkip;
TLMaybeComment:
 # '#' followed by a hex digit is a literal, not a comment.
$tmp = $scan + 1;
 goto TLDispatch  if $tmp >=u $send;
$next = [$tmp].bu;
$tmp =  call IsHexDigit($next);
 goto TLDispatch  if $tmp  is  nonzero;
 # Consume comment up to but not including the terminating newline.
TLComLoop:
 goto TLSkip  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - 10;
 goto TLSkip  if $tmp  is  zero;
$scan = $scan + 1;
 goto TLComLoop;
```

The dispatch phase is entered with `$scan` pointing at the first byte of a token. Before classifying the byte we capture `$tok_start`, `$tok_line`, and `$tok_col`, because several operators consume up to three more bytes and the parser must report errors against the start of the lexeme rather than its end.

⟨ ***Tokenize: capture token position*** ⟩ ≡

```
TLDispatch:
$tok_start = $scan;
$tok_line = $line;
$tmp = $scan - $line_start;
$tok_col = $tmp + 1;
$ch = [$scan].bu;
```

Single-character punctuation is handled first because it is the most common case and the cheapest to emit. Each arm emits its token and jumps back to `TLMain`; no arm falls through to the next.

The ampersand is deliberately one-character-only here: it is used both as the geta prefix (`&Label`) and as the bitwise-AND operator, and the parser disambiguates by context.

⟨ ***Tokenize: punctuation dispatch*** ⟩ ≡

```
$tmp = $ch - '(';
 goto TLP_LPAREN  if $tmp  is  zero;
$tmp = $ch - ')';
 goto TLP_RPAREN  if $tmp  is  zero;


$tmp = $ch - '[';
 goto TLP_LBRACKET  if $tmp  is  zero;
$tmp = $ch - ']';
 goto TLP_RBRACKET  if $tmp  is  zero;
$tmp = $ch - ',';
 goto TLP_COMMA  if $tmp  is  zero;
$tmp = $ch - ';';
 goto TLP_SEMI  if $tmp  is  zero;
$tmp = $ch - ':';
 goto TLP_COLON  if $tmp  is  zero;
$tmp = $ch - '&';
 goto TLP_AMP  if $tmp  is  zero;
$tmp = $ch - '|';
 goto TLP_PIPE  if $tmp  is  zero;
$tmp = $ch - '^';
 goto TLP_CARET  if $tmp  is  zero;
 goto TLOps;
TLP_LPAREN:
$scan = $scan + 1;
 call EmitToken(TK_LPAREN, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLP_RPAREN:
$scan = $scan + 1;
 call EmitToken(TK_RPAREN, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLP_LBRACKET:
$scan = $scan + 1;
 call EmitToken(TK_LBRACKET, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLP_RBRACKET:
$scan = $scan + 1;
 call EmitToken(TK_RBRACKET, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLP_COMMA:
$scan = $scan + 1;
 call EmitToken(TK_COMMA, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLP_SEMI:
$scan = $scan + 1;
 call EmitToken(TK_SEMI, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLP_COLON:
$scan = $scan + 1;
 call EmitToken(TK_COLON, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLP_AMP:
$scan = $scan + 1;
 call EmitToken(TK_AMP, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLP_PIPE:
$scan = $scan + 1;
```

```
 call EmitToken(TK_PIPE, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLP_CARET:
$scan = $scan + 1;
 call EmitToken(TK_CARET, 0, 0, $tok_line, $tok_col);
 goto TLMain;
```

Compound operators share the pattern: a base character (`+`, `-`, `*`, `/`, `%`, `=`, `!`, `<`, `>`, `?`) optionally followed by one or two modifier bytes. The arms below handle them from simplest to most complex. Arms that fall through to the single-byte emission share a common tail at the base label (for example `TLOpPlus` after checking for `+u` and `+.`).

The three-byte forms (`«u`, `<=u`, `<=.`, `>=u`, `>=.`) are handled by nested dispatch: after matching `«`, `<=`, `»`, or `>=`, we peek one more byte for the optional `u` or `.`.

⟨ ***Tokenize: operator dispatch*** ⟩ ≡

```
TLOps:
$tmp = $ch - '+';
 goto TLOpPlus  if $tmp  is  zero;
$tmp = $ch - '-';


 goto TLOpMinus  if $tmp  is  zero;
$tmp = $ch - '*';
 goto TLOpStar  if $tmp  is  zero;
$tmp = $ch - '/';
 goto TLOpSlash  if $tmp  is  zero;
$tmp = $ch - '%';
 goto TLOpPercent  if $tmp  is  zero;
$tmp = $ch - '=';
 goto TLOpEquals  if $tmp  is  zero;
$tmp = $ch - '!';
 goto TLOpBang  if $tmp  is  zero;
$tmp = $ch - '<';
 goto TLOpLT  if $tmp  is  zero;
$tmp = $ch - '>';
 goto TLOpGT  if $tmp  is  zero;
$tmp = $ch - '?';
 goto TLOpQuery  if $tmp  is  zero;
 goto TLScanners;

 # '+' then optional 'u' or '.'
TLOpPlus:
$scan = $scan + 1;
 goto TLPlusEmit  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - 'u';
 goto TLPlusU  if $tmp  is  zero;
$tmp = $ch - '.';
 goto TLPlusF  if $tmp  is  zero;
TLPlusEmit:
 call EmitToken(TK_PLUS, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLPlusU:
$scan = $scan + 1;
 call EmitToken(TK_PLUSU, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLPlusF:
$scan = $scan + 1;
```

```
 call EmitToken(TK_FADD, 0, 0, $tok_line, $tok_col);
 goto TLMain;

 # '-' then optional 'u', '.', or '>'
TLOpMinus:
$scan = $scan + 1;
 goto TLMinusEmit  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - 'u';
 goto TLMinusU  if $tmp  is  zero;
$tmp = $ch - '.';
 goto TLMinusF  if $tmp  is  zero;
$tmp = $ch - '>';
 goto TLMinusArrow  if $tmp  is  zero;
TLMinusEmit:
 call EmitToken(TK_MINUS, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLMinusU:
$scan = $scan + 1;
 call EmitToken(TK_MINUSU, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLMinusF:
$scan = $scan + 1;
 call EmitToken(TK_FSUB, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLMinusArrow:
$scan = $scan + 1;
 call EmitToken(TK_ARROW, 0, 0, $tok_line, $tok_col);
 goto TLMain;

 # '*' then optional 'u', '.$', or '.$u' style suffix.
TLOpStar:
$scan = $scan + 1;
 goto TLStarEmit  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - 'u';
 goto TLStarU  if $tmp  is  zero;
$tmp = $ch - '.';
 goto TLStarF  if $tmp  is  zero;
$tmp = $ch - '$';
 goto TLStarDollar  if $tmp  is  zero;
TLStarEmit:
 call EmitToken(TK_STAR, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLStarU:
$scan = $scan + 1;
 call EmitToken(TK_STARU, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLStarF:
$scan = $scan + 1;
 call EmitToken(TK_FMUL, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLStarDollar:
$tmp = $scan + 1;
 goto TLStarEmit  if $tmp >=u $send;
$ch = [$tmp].bu;
$tmp = $ch - 'u';
 goto TLStarEmit  if $tmp  is  nonzero;
$scan = $scan + 2;
 call EmitToken(TK_STARU, 0, 0, $tok_line, $tok_col);
 goto TLMain;
```

```
 # '/' then optional 'u', '.$', or '.$u' style suffix.
TLOpSlash:
$scan = $scan + 1;
 goto TLSlashEmit  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - 'u';
 goto TLSlashU  if $tmp  is  zero;
$tmp = $ch - '.';
 goto TLSlashF  if $tmp  is  zero;
$tmp = $ch - '$';
 goto TLSlashDollar  if $tmp  is  zero;
TLSlashEmit:
 call EmitToken(TK_SLASH, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLSlashU:
$scan = $scan + 1;
 call EmitToken(TK_SLASHU, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLSlashF:
$scan = $scan + 1;
 call EmitToken(TK_FDIV, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLSlashDollar:
$tmp = $scan + 1;
 goto TLSlashEmit  if $tmp >=u $send;
$ch = [$tmp].bu;
$tmp = $ch - 'u';
 goto TLSlashEmit  if $tmp  is  nonzero;
$scan = $scan + 2;
 call EmitToken(TK_SLASHU, 0, 0, $tok_line, $tok_col);
 goto TLMain;

 # '%' then optional '.'
TLOpPercent:
$scan = $scan + 1;
 goto TLPctEmit  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - '.';
 goto TLPctF  if $tmp  is  zero;
TLPctEmit:
 call EmitToken(TK_PERCENT, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLPctF:
$scan = $scan + 1;
 call EmitToken(TK_FREM, 0, 0, $tok_line, $tok_col);
 goto TLMain;

 # '=' then optional '='; the '==.' form is handled as a trailing '.'.
TLOpEquals:
$scan = $scan + 1;
 goto TLEqEmit  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - '=';
 goto TLEqEq  if $tmp  is  zero;
TLEqEmit:
 call EmitToken(TK_EQUALS, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLEqEq:
$scan = $scan + 1;
 goto TLEqEqEmit  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - '.';
```

```
 goto TLEqEqF  if $tmp  is  zero;
TLEqEqEmit:
 call EmitToken(TK_EQ, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLEqEqF:
$scan = $scan + 1;
 call EmitToken(TK_EQF, 0, 0, $tok_line, $tok_col);
 goto TLMain;

 # '!' must be followed by '='; the '!=.' form is a trailing '.'.
TLOpBang:
$scan = $scan + 1;
 goto TLBangErr  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - '=';
 goto TLBangEq  if $tmp  is  zero;
TLBangErr:
 call ErrHaltAt(&ErrLexChar, $tok_line, $tok_col);
TLBangEq:
$scan = $scan + 1;
 goto TLBangEqEmit  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - '.';
 goto TLBangEqF  if $tmp  is  zero;
TLBangEqEmit:
 call EmitToken(TK_NEQ, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLBangEqF:
$scan = $scan + 1;
 call EmitToken(TK_NEQF, 0, 0, $tok_line, $tok_col);
 goto TLMain;

 # '<' with all its follow-ons: <<, <<u, <=, <=u, <=., <u, <., <
TLOpLT:
$scan = $scan + 1;
 goto TLLTEmit  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - '<';
 goto TLLTLT  if $tmp  is  zero;
$tmp = $ch - '=';
 goto TLLTEq  if $tmp  is  zero;
$tmp = $ch - 'u';
 goto TLLTU  if $tmp  is  zero;
$tmp = $ch - '.';
 goto TLLTF  if $tmp  is  zero;
TLLTEmit:
 call EmitToken(TK_LT, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLLTLT:
$scan = $scan + 1;
 goto TLLTLTEmit  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - 'u';
 goto TLLTLTU  if $tmp  is  zero;
TLLTLTEmit:
 call EmitToken(TK_SHL, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLLTLTU:
$scan = $scan + 1;
 call EmitToken(TK_SHLU, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLLTEq:
```

```
$scan = $scan + 1;
 goto TLLTEqEmit  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - 'u';
 goto TLLTEqU  if $tmp  is  zero;
$tmp = $ch - '.';
 goto TLLTEqF  if $tmp  is  zero;
TLLTEqEmit:
 call EmitToken(TK_LEQ, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLLTEqU:
$scan = $scan + 1;
 call EmitToken(TK_LEQU, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLLTEqF:
$scan = $scan + 1;
 call EmitToken(TK_LEQF, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLLTU:
$scan = $scan + 1;
 call EmitToken(TK_LTU, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLLTF:
$scan = $scan + 1;
 call EmitToken(TK_LTF, 0, 0, $tok_line, $tok_col);
 goto TLMain;

 # '>' symmetric to '<'.
TLOpGT:
$scan = $scan + 1;
 goto TLGTEmit  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - '>';
 goto TLGTGT  if $tmp  is  zero;
$tmp = $ch - '=';
 goto TLGTEq  if $tmp  is  zero;
$tmp = $ch - 'u';
 goto TLGTU  if $tmp  is  zero;
$tmp = $ch - '.';
 goto TLGTF  if $tmp  is  zero;
TLGTEmit:
 call EmitToken(TK_GT, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLGTGT:
$scan = $scan + 1;
 goto TLGTGTEmit  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - 'u';
 goto TLGTGTU  if $tmp  is  zero;
TLGTGTEmit:
 call EmitToken(TK_SHR, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLGTGTU:
$scan = $scan + 1;
 call EmitToken(TK_SHRU, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLGTEq:
$scan = $scan + 1;
 goto TLGTEqEmit  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - 'u';
 goto TLGTEqU  if $tmp  is  zero;
```

```
    $tmp = $ch - '.';
     goto TLGTEqF  if $tmp  is  zero;
    TLGTEqEmit:
     call EmitToken(TK_GEQ, 0, 0, $tok_line, $tok_col);
     goto TLMain;
    TLGTEqU:
    $scan = $scan + 1;
     call EmitToken(TK_GEQU, 0, 0, $tok_line, $tok_col);
     goto TLMain;
    TLGTEqF:
    $scan = $scan + 1;
     call EmitToken(TK_GEQF, 0, 0, $tok_line, $tok_col);
     goto TLMain;
    TLGTU:
    $scan = $scan + 1;
     call EmitToken(TK_GTU, 0, 0, $tok_line, $tok_col);
     goto TLMain;
    TLGTF:
    $scan = $scan + 1;
     call EmitToken(TK_GTF, 0, 0, $tok_line, $tok_col);
     goto TLMain;

     # '?' must be followed by '='.
    TLOpQuery:
    $scan = $scan + 1;
     goto TLQueryErr  if $scan >=u $send;
    $ch = [$scan].bu;
    $tmp = $ch - '=';
     goto TLQueryEq  if $tmp  is  zero;
    TLQueryErr:
     call ErrHaltAt(&ErrLexChar, $tok_line, $tok_col);
    TLQueryEq:
    $scan = $scan + 1;
     call EmitToken(TK_ZASSIGN, 0, 0, $tok_line, $tok_col);
     goto TLMain;
```

12. Deferred scanners: identifiers and keywords.

The first byte of the unclassified lexeme is already in `$ch`. If it is a letter or underscore, we scan the identifier body, look the result up in `KeywordTable`, and emit either `TK_KW` (with the keyword index as `TK_VALUE`) or `TK_LABEL` (with a `StringPool` offset). Every other non-ASCII-punctuation start byte—digits, `$`, `'`, `"`, `.`, `#`—is dispatched by the surrounding lexer loop to its dedicated scanner branch.

The identifier body is `[A-Za-z0-9_]*`; the scan stops at the first byte that fails `IsIdentCont`. The length passed to `EmitToken` and `PoolIntern` is `$scan - $tok_start`, computed after the loop. For keyword tokens `TK_LEN` is still set to the lexeme length so that diagnostics can echo the source spelling, even though the parser will normally dispatch on `TK_VALUE`.

⟨ ***Tokenize: deferred scanners*** ⟩ ≡

```
TLScanners:
$tmp = call IsIdentStart($ch);
 goto TLScanIdent if $tmp is nonzero;
$tmp = $ch - ' #';


 goto TLScanHex if $tmp is zero;
$tmp = call IsDigit($ch);
 goto TLScanNum if $tmp is nonzero;
$tmp = $ch - 34;
 goto TLScanStr if $tmp is zero;
$tmp = $ch - 39;
 goto TLScanChar if $tmp is zero;
$tmp = $ch - '$';
 goto TLScanDollar if $tmp is zero;
$tmp = $ch - '.';
 goto TLScanDot if $tmp is zero;
 # Nothing matched: fatal lexical error.
 call ErrHaltAt(&ErrLexChar, $tok_line, $tok_col);

TLScanIdent:
 # Advance past identifier body.
TLScanIdentLoop:
$scan = $scan + 1;
 goto TLScanIdentDone if $scan >=u $send;
$ch = [$scan].bu;
$tmp = call IsIdentCont($ch);
 goto TLScanIdentLoop if $tmp is nonzero;
TLScanIdentDone:
 # Peek next byte to detect suffix-aware forms: ident+'.' → unary-op or
 # dotted-keyword; ident+'=' → wyde-op. If either post-check matches,
 # we consume the suffix byte and emit the specialised token. If not,
 # control falls through to plain keyword/label classification.
$idlen = $scan - $tok_start;
 goto TLScanIdentPlain if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - '.';
 goto TLScanIdentDot if $tmp is zero;
$tmp = $ch - '=';
 goto TLScanIdentEq if $tmp is zero;
TLScanIdentPlain:
$tmp = call KeywordLookup($tok_start, $idlen);
 goto TLScanIdentLabel if $tmp is zero;
$idkind = TK_KW;
$idvalue = $tmp;
 goto TLScanIdentEmit;
TLScanIdentLabel:
$idkind = TK_LABEL;
$idvalue = call PoolIntern($tok_start, $idlen);
 goto TLScanIdentEmit;
TLScanIdentDot:
$idvalue = &UnaryOpTable;
$tmp = call TableLookup($idvalue, $tok_start, $idlen);
 goto TLScanIdentDotUnary if $tmp is nonzero;
$idvalue = call SregStyleLookup(&DotKwTable, $tok_start, $idlen);
 goto TLScanIdentDotKw if $idvalue is nonzero;
 goto TLScanIdentPlain;
TLScanIdentDotUnary:
$scan = $scan + 1;                    # consume '.'
$idlen = $scan - $tok_start;
$idkind = TK_UNARYOP;
```

```
$idvalue = $tmp;
 goto TLScanIdentEmit;
TLScanIdentDotKw:
$scan = $scan + 1;                        # consume '.'
$idlen = $scan - $tok_start;
$idkind = TK_KW;
$idvalue = $idvalue - 1;                 # undo +1 bias from SregStyleLookup
 goto TLScanIdentEmit;
TLScanIdentEq:
$idvalue = &WydeOpTable;
$tmp =  call TableLookup($idvalue, $tok_start, $idlen);
 goto TLScanIdentPlain  if $tmp  is  zero;
$scan = $scan + 1;                        # consume '='
$idlen = $scan - $tok_start;
$idkind = TK_WYDEOP;
$idvalue = $tmp;
TLScanIdentEmit:
 call EmitToken($idkind, $idvalue, $idlen, $tok_line, $tok_col);
 goto TLMain;
```

13. Numeric literals and scaled-add tokens.

Decimal integers match `[0-9]+` and emit `TK_IMMEDIATE` with the value accumulated in `$numval`. Float literals match `[0-9]+\.[0-9]+([eE][+-]?[0-9]+)?` and emit `TK_FLOAT`; their raw text is interned in `StringPool` and the parser converts it to a bit pattern later. Hex literals match `#[0-9A-Fa-f]+` and emit `TK_IMMEDIATE` accumulated via shift-or.

Scaled-add tokens `2addu`, `4addu`, `8addu`, and `16addu` emit `TK_nADDU`. Detection requires matching the four-byte suffix `addu` with a word boundary immediately after it, so that e.g. `164addu` tokenises as the integer `164` followed by the identifier `addu`. This check must happen before the float check so that `2addu` is never misread as the integer `2` followed by the identifier `addu`.

⟨ ***Tokenize: deferred scanners*** ⟩ +≡

```
TLScanNum:
$numval = 0;
TLScanNumLoop:
$ch = [$scan].bu;


$tmp =  call IsDigit($ch);
 goto TLScanNumDone  if $tmp  is  zero;
$numval = $numval * 10;
$tmp = $ch - '0';
$numval = $numval + $tmp;
$scan = $scan + 1;
 goto TLScanNumDone  if $scan >=u $send;
 goto TLScanNumLoop;
TLScanNumDone:
 # Peek for "addu" suffix.  The digit length must be 1 (for 2/4/8) or 2
 # (for 16) to be a candidate, and four bytes 'a','d','d','u' must follow
 # with no further ident-cont byte after them.
$idlen = $scan - $tok_start;
$tmp = $idlen - 2;
 goto TLScanNumMaybeFloat  if $tmp  is  pos;
$tmp = $scan + 4;
 goto TLScanNumMaybeFloat  if $tmp >u $send;
$ch = [$scan].bu;
```

```
$tmp = $ch - 'a';
 goto TLScanNumMaybeFloat  if $tmp  is  nonzero;
$tmp = $scan + 1;
$ch = [$tmp].bu;
$tmp = $ch - 'd';
 goto TLScanNumMaybeFloat  if $tmp  is  nonzero;
$tmp = $scan + 2;
$ch = [$tmp].bu;
$tmp = $ch - 'd';
 goto TLScanNumMaybeFloat  if $tmp  is  nonzero;
$tmp = $scan + 3;
$ch = [$tmp].bu;
$tmp = $ch - 'u';
 goto TLScanNumMaybeFloat  if $tmp  is  nonzero;
$tmp = $scan + 4;
 goto TLScanNumAdduBoundaryOK  if $tmp >=u $send;
$ch = [$tmp].bu;
$tmp =  call IsIdentCont($ch);
 goto TLScanNumMaybeFloat  if $tmp  is  nonzero;
TLScanNumAdduBoundaryOK:
$tmp = $idlen - 1;
 goto TLScanNumAddu2digit  if $tmp  is  nonzero;
 # One-digit scaled-add: must be 2, 4, or 8.
$ch = [$tok_start].bu;
$tmp = $ch - '2';
 goto TLScanNumEmit2  if $tmp  is  zero;
$tmp = $ch - '4';
 goto TLScanNumEmit4  if $tmp  is  zero;
$tmp = $ch - '8';
 goto TLScanNumEmit8  if $tmp  is  zero;
 goto TLScanNumMaybeFloat;
TLScanNumAddu2digit:
 # Two-digit scaled-add: must be exactly "16".
$ch = [$tok_start].bu;
$tmp = $ch - '1';
 goto TLScanNumMaybeFloat  if $tmp  is  nonzero;
$tmp = $tok_start + 1;
$ch = [$tmp].bu;
$tmp = $ch - '6';
 goto TLScanNumMaybeFloat  if $tmp  is  nonzero;
$scan = $scan + 4;
 call EmitToken(TK_16ADDU, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLScanNumEmit2:
$scan = $scan + 4;
 call EmitToken(TK_2ADDU, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLScanNumEmit4:
$scan = $scan + 4;
 call EmitToken(TK_4ADDU, 0, 0, $tok_line, $tok_col);
 goto TLMain;
TLScanNumEmit8:
$scan = $scan + 4;
 call EmitToken(TK_8ADDU, 0, 0, $tok_line, $tok_col);
 goto TLMain;

TLScanNumMaybeFloat:
 # A dot immediately followed by a digit promotes the integer to a float.
 # Anything else (including end-of-buffer or a dotted size suffix without
 # a trailing digit, which can't occur on an integer anyway) leaves the
 # integer intact.
 goto TLScanNumIntEmit  if $scan >=u $send;
```

```
$ch = [$scan].bu;
$tmp = $ch - '.';
 goto TLScanNumIntEmit  if $tmp  is  nonzero;
$tmp = $scan + 1;
 goto TLScanNumIntEmit  if $tmp >=u $send;
$ch = [$tmp].bu;
$tmp =  call IsDigit($ch);
 goto TLScanNumIntEmit  if $tmp  is  zero;
 goto TLScanNumFloat;

TLScanNumIntEmit:
$idlen = $scan - $tok_start;
 call EmitToken(TK_IMMEDIATE, $numval, $idlen, $tok_line, $tok_col);
 goto TLMain;

TLScanNumFloat:
$scan = $scan + 1;      # consume '.'
TLScanNumFloatFrac:
 goto TLScanNumFloatExpCheck  if $scan >=u $send;
$ch = [$scan].bu;
$tmp =  call IsDigit($ch);
 goto TLScanNumFloatExpCheck  if $tmp  is  zero;
$scan = $scan + 1;
 goto TLScanNumFloatFrac;
TLScanNumFloatExpCheck:
 goto TLScanNumFloatEmit  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - 'e';
 goto TLScanNumFloatExp  if $tmp  is  zero;
$tmp = $ch - 'E';
 goto TLScanNumFloatExp  if $tmp  is  zero;
 goto TLScanNumFloatEmit;
TLScanNumFloatExp:
$scan = $scan + 1;
 goto TLScanNumFloatExpDigits  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - '+';
 goto TLScanNumFloatExpSign  if $tmp  is  zero;
$tmp = $ch - '-';
 goto TLScanNumFloatExpSign  if $tmp  is  zero;
 goto TLScanNumFloatExpDigits;
TLScanNumFloatExpSign:
$scan = $scan + 1;
TLScanNumFloatExpDigits:
 goto TLScanNumFloatEmit  if $scan >=u $send;
$ch = [$scan].bu;
$tmp =  call IsDigit($ch);
 goto TLScanNumFloatEmit  if $tmp  is  zero;
$scan = $scan + 1;
 goto TLScanNumFloatExpDigits;
TLScanNumFloatEmit:
$idlen = $scan - $tok_start;
$idvalue =  call PoolIntern($tok_start, $idlen);
 call EmitToken(TK_FLOAT, $idvalue, $idlen, $tok_line, $tok_col);
 goto TLMain;

TLScanHex:
 # '#' was confirmed to be followed by a hex digit by the skip phase,
 # so we consume the '#' and at least one hex digit unconditionally.
$scan = $scan + 1;
$numval = 0;
TLScanHexLoop:
```

```
 goto TLScanHexDone  if $scan >=u $send;
$ch = [$scan].bu;
$tmp =  call IsHexDigit($ch);
 goto TLScanHexDone  if $tmp  is  zero;
$numval = $numval << 4;
$tmp =  call HexDigitValue($ch);
$numval = $numval | $tmp;
$scan = $scan + 1;
 goto TLScanHexLoop;
TLScanHexDone:
$idlen = $scan - $tok_start;
 call EmitToken(TK_IMMEDIATE, $numval, $idlen, $tok_line, $tok_col);
 goto TLMain;
```

14. String and character literals.

String literals `"..."` decode their escape sequences and append the resulting bytes directly into `StringPool`; `TK_VALUE` holds the start offset of the decoded content and `TK_LEN` holds the byte count. Character literals `'c'` or `'\n'` emit `TK_IMMEDIATE` with `TK_VALUE` set to the Unicode code point (0–255 for plain bytes; `\u` and `\U` escapes are also handled here). Both forms recognise the standard C escapes `\n`, `\t`, `\r`, `\0`, `\\`, `\"`, `\'`, `\a`, `\b`, `\f`, and `\v` through `DecodeEscape`. An unterminated literal or a bare newline inside one is a fatal error (`ErrLexUnterm`); an unknown escape byte raises `ErrLexEscape`.

⟨ ***Tokenize: deferred scanners*** ⟩ +≡

```
TLScanStr:
$scan = $scan + 1;                # consume opening "
$idvalue = $str_end;              # pool offset of decoded bytes
TLScanStrLoop:


 goto TLScanStrUnterm  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - 34;
 goto TLScanStrEnd  if $tmp  is  zero;
$tmp = $ch - 92;
 goto TLScanStrEscape  if $tmp  is  zero;
$tmp = $ch - '\n';
 goto TLScanStrUnterm  if $tmp  is  zero;
 call PoolAppendByte($ch);
$scan = $scan + 1;
 goto TLScanStrLoop;
TLScanStrEscape:
$scan = $scan + 1;
 goto TLScanStrUnterm  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - 'u';
 goto TLScanStrEscU4  if $tmp  is  zero;
$tmp = $ch - 'U';
 goto TLScanStrEscU8  if $tmp  is  zero;
$tmp =  call DecodeEscape($ch, $tok_line, $tok_col);
 call PoolAppendByte($tmp);
$scan = $scan + 1;
 goto TLScanStrLoop;
TLScanStrEscU4:
$numval = 0;
$next = 4;
```

```
 goto TLScanStrEscHex;
TLScanStrEscU8:
$numval = 0;
$next = 8;
TLScanStrEscHex:
$scan = $scan + 1;
 goto TLScanStrEscBad  if $scan >=u $send;
$ch = [$scan].bu;
$tmp =  call IsHexDigit($ch);
 goto TLScanStrEscBad  if $tmp  is  zero;
$numval = $numval << 4;
$tmp =  call HexDigitValue($ch);
$numval = $numval | $tmp;
$next = $next - 1;
 goto TLScanStrEscHex  if $next  is  nonzero;
$tmp = $numval - #80;
 goto TLScanStrEsc1  if $tmp  is  neg;
$tmp = $numval - #800;
 goto TLScanStrEsc2  if $tmp  is  neg;
$tmp = $numval - #10000;
 goto TLScanStrEsc3  if $tmp  is  neg;
$tmp = $numval - #110000;
 goto TLScanStrEscBad  if $tmp  is  nonneg;
 goto TLScanStrEsc4;
TLScanStrEsc1:
 call PoolAppendByte($numval);
 goto TLScanStrEscDone;
TLScanStrEsc2:
$idkind = $numval >>u 6;
$idkind = $idkind | #C0;
 call PoolAppendByte($idkind);
$idkind = $numval & #3F;
$idkind = $idkind | #80;
 call PoolAppendByte($idkind);
 goto TLScanStrEscDone;
TLScanStrEsc3:
$idkind = $numval >>u 12;
$idkind = $idkind | #E0;
 call PoolAppendByte($idkind);
$idkind = $numval >>u 6;
$idkind = $idkind & #3F;
$idkind = $idkind | #80;
 call PoolAppendByte($idkind);
$idkind = $numval & #3F;
$idkind = $idkind | #80;
 call PoolAppendByte($idkind);
 goto TLScanStrEscDone;
TLScanStrEsc4:
$idkind = $numval >>u 18;
$idkind = $idkind | #F0;
 call PoolAppendByte($idkind);
$idkind = $numval >>u 12;
$idkind = $idkind & #3F;
$idkind = $idkind | #80;
 call PoolAppendByte($idkind);
$idkind = $numval >>u 6;
$idkind = $idkind & #3F;
$idkind = $idkind | #80;
 call PoolAppendByte($idkind);
$idkind = $numval & #3F;
$idkind = $idkind | #80;
 call PoolAppendByte($idkind);
```

```
TLScanStrEscDone:
$scan = $scan + 1;
 goto TLScanStrLoop;
TLScanStrEscBad:
 call ErrHaltAt(&ErrLexEscape, $tok_line, $tok_col);
TLScanStrEnd:
$scan = $scan + 1;                # consume closing "
$idlen = $str_end - $idvalue;
 call EmitToken(TK_STRING, $idvalue, $idlen, $tok_line, $tok_col);
 goto TLMain;
TLScanStrUnterm:
 call ErrHaltAt(&ErrLexUnterm, $tok_line, $tok_col);

TLScanChar:
$scan = $scan + 1;                # consume opening '
 goto TLScanCharUnterm  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - 39;
 goto TLScanCharUnterm  if $tmp  is  zero;    # empty literal
$tmp = $ch - '\n';
 goto TLScanCharUnterm  if $tmp  is  zero;
$tmp = $ch - 92;
 goto TLScanCharEscape  if $tmp  is  zero;
$tmp = $ch  cmpu 128;
 goto TLScanCharUtf8  if $tmp  is  nonneg;
$numval = $ch;
$scan = $scan + 1;
 goto TLScanCharClose;
TLScanCharUtf8:
 # Decode one UTF-8 code point directly in the literal body.
$tmp = $ch & #E0;
$tmp = $tmp - #C0;
 goto TLScanCharUtf8_2  if $tmp  is  zero;
$tmp = $ch & #F0;
$tmp = $tmp - #E0;
 goto TLScanCharUtf8_3  if $tmp  is  zero;
$tmp = $ch & #F8;
$tmp = $tmp - #F0;
 goto TLScanCharUtf8_4  if $tmp  is  zero;
 goto TLScanCharEscBad;
TLScanCharUtf8_2:
$numval = $ch & #1F;
$next = 1;
 goto TLScanCharUtf8Loop;
TLScanCharUtf8_3:
$numval = $ch & #0F;
$next = 2;
 goto TLScanCharUtf8Loop;
TLScanCharUtf8_4:
$numval = $ch & #07;
$next = 3;
TLScanCharUtf8Loop:
$scan = $scan + 1;
 goto TLScanCharUnterm  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch & #C0;
$tmp = $tmp - #80;
 goto TLScanCharEscBad  if $tmp  is  nonzero;
$numval = $numval << 6;
$tmp = $ch & #3F;
$numval = $numval | $tmp;
$next = $next - 1;
```

```
 goto TLScanCharUtf8Loop  if $next  is  nonzero;
$scan = $scan + 1;
 goto TLScanCharClose;
TLScanCharEscape:
$scan = $scan + 1;
 goto TLScanCharUnterm  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - 'u';
 goto TLScanCharEscU4  if $tmp  is  zero;
$tmp = $ch - 'U';
 goto TLScanCharEscU8  if $tmp  is  zero;
$numval =  call DecodeEscape($ch, $tok_line, $tok_col);
$scan = $scan + 1;
 goto TLScanCharClose;
TLScanCharEscU4:
$numval = 0;
$next = 4;
 goto TLScanCharEscHex;
TLScanCharEscU8:
$numval = 0;
$next = 8;
TLScanCharEscHex:
$scan = $scan + 1;
 goto TLScanCharEscBad  if $scan >=u $send;
$ch = [$scan].bu;
$tmp =  call IsHexDigit($ch);
 goto TLScanCharEscBad  if $tmp  is  zero;
$numval = $numval << 4;
$tmp =  call HexDigitValue($ch);
$numval = $numval | $tmp;
$next = $next - 1;
 goto TLScanCharEscHex  if $next  is  nonzero;
$tmp = $numval - #110000;
 goto TLScanCharEscBad  if $tmp  is  nonneg;
$scan = $scan + 1;
 goto TLScanCharClose;
TLScanCharEscBad:
 call ErrHaltAt(&ErrLexEscape, $tok_line, $tok_col);
TLScanCharClose:
 goto TLScanCharUnterm  if $scan >=u $send;
$ch = [$scan].bu;
$tmp = $ch - 39;
 goto TLScanCharUnterm  if $tmp  is  nonzero;
$scan = $scan + 1;
$idlen = $scan - $tok_start;
 call EmitToken(TK_IMMEDIATE, $numval, $idlen, $tok_line, $tok_col);
 goto TLMain;
TLScanCharUnterm:
 call ErrHaltAt(&ErrLexUnterm, $tok_line, $tok_col);
```

15. Dollar-prefixed terminals.

`$N` emits `TK_REGISTER` with `TK_VALUE` set to `N`. `$rA`–`$rZZ` emit `TK_SREG` with the MMIX special-register index (0–31) as `TK_VALUE`. Any other `$name` emits `TK_REGNAME` with `TK_VALUE` set to the pool offset of the full lexeme including the `$`.

The dispatch consumes `$`, then inspects the next byte: a digit triggers the register-number scan; an ident-start byte triggers an identifier scan whose result is post-classified as `TK_SREG` or

TK_REGNAME depending on whether it matches an entry in `SregTable`. A bare `$` with no syntactic follower is a lexical error, ensuring every emitted dollar-token carries a non-empty payload.

⟨ ***Tokenize: deferred scanners*** ⟩ +≡

```
TLScanDollar:
$scan = $scan + 1;                  # consume '$'
 goto TLScanDollarErr  if $scan >=u $send;
$ch = [$scan].bu;


$tmp =  call IsDigit($ch);
 goto TLScanDollarReg  if $tmp  is  nonzero;
$tmp =  call IsIdentStart($ch);
 goto TLScanDollarName  if $tmp  is  nonzero;
TLScanDollarErr:
 call ErrHaltAt(&ErrLexChar, $tok_line, $tok_col);

TLScanDollarReg:
$numval = 0;
TLScanDollarRegLoop:
$ch = [$scan].bu;
$tmp =  call IsDigit($ch);
 goto TLScanDollarRegDone  if $tmp  is  zero;
$numval = $numval * 10;
$tmp = $ch - '0';
$numval = $numval + $tmp;
$scan = $scan + 1;
 goto TLScanDollarRegDone  if $scan >=u $send;
 goto TLScanDollarRegLoop;
TLScanDollarRegDone:
$idlen = $scan - $tok_start;
 call EmitToken(TK_REGISTER, $numval, $idlen, $tok_line, $tok_col);
 goto TLMain;

TLScanDollarName:
$name_start = $scan;
TLScanDollarNameLoop:
$scan = $scan + 1;
 goto TLScanDollarNameDone  if $scan >=u $send;
$ch = [$scan].bu;
$tmp =  call IsIdentCont($ch);
 goto TLScanDollarNameLoop  if $tmp  is  nonzero;
TLScanDollarNameDone:
 # SREG candidate: first letter must be 'r' and suffix non-empty.
$ch = [$name_start].bu;
$tmp = $ch - 'r';
 goto TLScanDollarRegname  if $tmp  is  nonzero;
$idlen = $scan - $name_start;
$tmp = $idlen - 1;
 goto TLScanDollarRegname  if $tmp  is  nonpos;
$tmp = $name_start + 1;
$idvalue = $idlen - 1;
$numval =  call SregLookup($tmp, $idvalue);
 goto TLScanDollarRegname  if $numval  is  zero;
$numval = $numval - 1;              # undo +1 bias returned by SregLookup
$idlen = $scan - $tok_start;
 call EmitToken(TK_SREG, $numval, $idlen, $tok_line, $tok_col);
 goto TLMain;

TLScanDollarRegname:
$idlen = $scan - $tok_start;
```

```
$idvalue = call PoolIntern($tok_start, $idlen);
 call EmitToken(TK_REGNAME, $idvalue, $idlen, $tok_line, $tok_col);
 goto TLMain;
```

16. Dot-prefixed size suffixes.

The suffixes `.b`, `.bu`, `.w`, `.wu`, `.t`, `.tu`, `.o`, `.sf`, `.ht`, and `.unc` each emit `TK_SIZE` with a code in the range 1–10. A lone `.` or `.` followed by an unrecognised name is a fatal lexer error, since every dot-prefixed lexeme in PL/MMIX is a size suffix. The other contexts in which `.` appears—as the decimal point in a float literal like `3.14`, or as the trailing dot in `sqrt.` or `cmpe.`—are handled in their respective scanners before `TLScanDot` is reached.

⟨ ***Tokenize: deferred scanners*** ⟩ +≡

```
TLScanDot:
$scan = $scan + 1;                    # consume '.'
$name_start = $scan;
TLScanDotLoop:
 goto TLScanDotDone if $scan >=u $send;
$ch = [$scan].bu;
$tmp = call IsIdentCont($ch);
 goto TLScanDotDone if $tmp is zero;
$scan = $scan + 1;
 goto TLScanDotLoop;
TLScanDotDone:
$idlen = $scan - $name_start;
 goto TLScanDotErr if $idlen is zero;
$idvalue = &SizeTable;
$numval = call TableLookup($idvalue, $name_start, $idlen);
 goto TLScanDotErr if $numval is zero;
$idlen = $scan - $tok_start;
 call EmitToken(TK_SIZE, $numval, $idlen, $tok_line, $tok_col);
 goto TLMain;
TLScanDotErr:
 call ErrHaltAt(&ErrLexChar, $tok_line, $tok_col);
```

A single `TK_EOF` token with a zero payload and the position just past the last byte of the source terminates the stream.

⟨ ***Tokenize: emit EOF*** ⟩ ≡

```
$tok_line = $line;
$tmp = $scan - $line_start;
$tok_col = $tmp + 1;
 call EmitToken(TK_EOF, 0, 0, $tok_line, $tok_col);
 return;
```

17. Token emission and helpers.

`EmitToken` appends one 32-byte record to `TokenBuf`. The line and column are packed into a single octa (line in the low 32 bits, column in the high 32 bits) so that the record layout declared in the constants section stays at four octas.

`IsHexDigit` returns 1 if its byte argument is a valid hexadecimal digit (`0`–`9`, `A`–`F`, `a`–`f`) and 0 otherwise. It is used both by the whitespace/comment skipper (to disambiguate `#` from a hex literal) and by the hex-literal scanner in the lexer section.

⟨ ***Lexer procedures*** ⟩ +≡

```
def EmitToken($kind, $value, $len, $line, $col):
  register $entry, $lc, $tmp;
 $tmp = $tok_cnt  cmpu TOKEN_MAX;
  if $tmp  is  nonneg:
    call ErrHaltAt("plmmix-compiler: too many tokens\n", $line, $col);
  end
 $entry = $tok_cnt << 5;              # * TOKEN_ENTRY_SIZE (= 32)
 $entry = $entry + $TokenBuf;
 [$entry + TK_KIND].o  = $kind;
 [$entry + TK_VALUE].o = $value;
 [$entry + TK_LEN].o   = $len;
 $lc = $col << 32;
 $lc = $lc | $line;
 [$entry + TK_LINE].o  = $lc;
 $tok_cnt = $tok_cnt + 1;
  return;
end
```

⟨ ***Lexer procedures*** ⟩ +≡

```
def IsHexDigit($ch) -> $r:
  register $tmp;
 $tmp = $ch - '0';
  goto IHDUpper  if $tmp  is  neg;
 $tmp = $ch - '9';
  if $tmp  is  nonpos:
    return 1;
  end
 IHDUpper:
 $tmp = $ch - 'A';
  goto IHDLower  if $tmp  is  neg;
 $tmp = $ch - 'F';
  if $tmp  is  nonpos:
    return 1;
  end
 IHDLower:
 $tmp = $ch - 'a';
  if $tmp  is  neg:
    return 0;
  end
 $tmp = $ch - 'f';
  if $tmp  is  nonpos:
    return 1;
  end
  return 0;
end
```

`DecodeEscape` takes the byte immediately following a backslash in a string or character literal and returns the decoded byte value. The recognised escapes are the standard C set: `\n`, `\t`, `\r`, `\0`, `\\`, `\"`, `\'`, `\a`, `\b`, `\f`, `\v`. Unicode escapes (`\uNNNN`, `\UNNNNNNNN`) are deferred until the compiler needs to parse its own sources using non-ASCII bytes, which it currently does not. Any

other follower byte raises `ErrLexEscape` via `ErrHalt`.

⟨ ***Lexer procedures*** ⟩ +≡

```
def DecodeEscape($ch, $line, $col) -> $r:
  register $tmp;
 $tmp = $ch - 'n';
  if $tmp  is  zero:


    return 10;
  end
 $tmp = $ch - 't';
  if $tmp  is  zero:
    return 9;
  end
 $tmp = $ch - 'r';
  if $tmp  is  zero:
    return 13;
  end
 $tmp = $ch - '0';
  if $tmp  is  zero:
    return 0;
  end
 $tmp = $ch - 92;
  if $tmp  is  zero:
    return 92;
  end
 $tmp = $ch - 34;
  if $tmp  is  zero:
    return 34;
  end
 $tmp = $ch - 39;
  if $tmp  is  zero:
    return 39;
  end
 $tmp = $ch - 'a';
  if $tmp  is  zero:
    return 7;
  end
 $tmp = $ch - 'b';
  if $tmp  is  zero:
    return 8;
  end
 $tmp = $ch - 'f';
  if $tmp  is  zero:
    return 12;
  end
 $tmp = $ch - 'v';
  if $tmp  is  zero:
    return 11;
  end
  call ErrHaltAt(&ErrLexEscape, $line, $col);
end
```

`IsDigit` recognises ASCII decimal digits `0`–`9`. `HexDigitValue` maps any ASCII hex digit (`0`–`9`, `A`–`F`, `a`–`f`) to its numeric value 0–15; callers must verify hex-ness first via `IsHexDigit`, as `HexDigitValue` does not validate its input.

⟨***Lexer procedures***⟩ +≡

```
def IsDigit($ch) -> $r:
  register $tmp;
 $tmp = $ch - '0';
  if $tmp  is  neg:
    return 0;
  end
 $tmp = $ch - '9';
  if $tmp  is  nonpos:
    return 1;
  end
  return 0;
end
```

⟨***Lexer procedures***⟩ +≡

```
def HexDigitValue($ch) -> $r:
  register $tmp;
 $tmp = $ch - '9';
  goto HDVNotDec  if $tmp  is  pos;
 $r = $ch - '0';
  return $r;
 HDVNotDec:
 $tmp = $ch - 'a';
  goto HDVLower  if $tmp  is  nonneg;
 $r = $ch - 'A';
 $r = $r + 10;
  return $r;
 HDVLower:
 $r = $ch - 'a';
 $r = $r + 10;
  return $r;
end
```

`IsIdentStart` recognises the first byte of a PL/MMIX identifier: an ASCII letter (`A`–`Z`, `a`–`z`), underscore, or any byte with the high bit set (>=128). This byte-oriented rule lets UTF-8 source names (for example Greek register aliases) pass through the bootstrap lexer without requiring full Unicode-category tables.

`IsIdentCont` additionally accepts decimal digits, which are permitted from the second byte onward, and inherits the >=128 allowance from `IsIdentStart`.

Both predicates return 1 or 0 in the output register so that the caller can dispatch with `if ... is nonzero`. `IsIdentCont` delegates its letter/underscore check to `IsIdentStart` to keep the two tables in sync.

⟨***Lexer procedures***⟩ +≡

```
def IsIdentStart($ch) -> $r:
  register $tmp;
 $tmp = $ch  cmpu 128;
  if $tmp  is  nonneg:
    return 1;
  end
 $tmp = $ch - '_';
  if $tmp  is  zero:
    return 1;
  end
 $tmp = $ch - 'A';
  goto IISLower  if $tmp  is  neg;
 $tmp = $ch - 'Z';
  if $tmp  is  nonpos:
    return 1;
  end
 IISLower:
 $tmp = $ch - 'a';
  if $tmp  is  neg:
    return 0;
  end
 $tmp = $ch - 'z';
  if $tmp  is  nonpos:
    return 1;
  end
  return 0;
end
```

⟨***Lexer procedures***⟩ +≡

```
def IsIdentCont($ch) -> $r:
  register $tmp;
 $r =  call IsIdentStart($ch);
  if $r  is  nonzero:
    return 1;
  end
 $tmp = $ch - '0';
  if $tmp  is  neg:
    return 0;
  end
 $tmp = $ch - '9';
  if $tmp  is  nonpos:
    return 1;
  end
  return 0;
end
```

`KeywordLookup` walks `KeywordTable` entry by entry, returning the 1-based index of the first entry whose length and bytes match the candidate identifier, or 0 if the table terminator is reached without a match. Each table entry is laid out as a one-byte length followed by that many payload bytes; a zero length byte terminates the table.

The comparison loop is short-circuit: a length mismatch skips directly to the next entry without touching the payload bytes, and a payload mismatch abandons the current entry at the first differing byte. With 74 keywords averaging five bytes, the expected cost per identifier is small

enough that a linear scan is competitive with a hash table, and the table needs no dynamic initialisation.

⟨ ***Lexer procedures*** ⟩ +≡

```
def KeywordLookupRange($tbl, $base_idx, $start, $len) -> $r:
  register $p, $entry_len, $tmp, $i, $sp, $kp, $first, $ch1, $ch2, $idx;
 $p = $tbl;
 $idx = $base_idx;


 $first = [$start].bu;
 KLLoop:
 $entry_len = [$p].bu;
  if $entry_len  is  zero:
    return 0;
  end
 $idx = $idx + 1;
 $p = $p + 1;
 $tmp = $entry_len - $len;
  goto KLNext  if $tmp  is  nonzero;
 $ch2 = [$p].bu;
 $tmp = $first - $ch2;
  goto KLNext  if $tmp  is  nonzero;
 $i = 1;
 KLCmp:
 $tmp = $i - $len;
  if $tmp  is  zero:
    return $idx;
  end
 $sp = $start + $i;
 $kp = $p + $i;
 $ch1 = [$sp].bu;
 $ch2 = [$kp].bu;
 $tmp = $ch1 - $ch2;
  goto KLNext  if $tmp  is  nonzero;
 $i = $i + 1;
  goto KLCmp;
 KLNext:
 $p = $p + $entry_len;
  goto KLLoop;
end
```

⟨ ***Lexer procedures*** ⟩ +≡

```
def KeywordLookup($start, $len) -> $r:
  register $ch, $tmp, $tbl;
 $r = 0;
 $ch = [$start].bu;


 $tmp = $ch - 'a';
  goto KLA  if $tmp  is  zero;
 $tmp = $ch - 'b';
  goto KLB  if $tmp  is  zero;
 $tmp = $ch - 'c';
  goto KLC  if $tmp  is  zero;
 $tmp = $ch - 'd';
  goto KLD  if $tmp  is  zero;
 $tmp = $ch - 'e';
  goto KLE  if $tmp  is  zero;
```

```
$tmp = $ch - 'g';
 goto KLG  if $tmp  is  zero;
$tmp = $ch - 'i';
 goto KLI  if $tmp  is  zero;
$tmp = $ch - 'l';
 goto KLL  if $tmp  is  zero;
$tmp = $ch - 'm';
 goto KLM  if $tmp  is  zero;
$tmp = $ch - 'n';
 goto KLN  if $tmp  is  zero;
$tmp = $ch - 'o';
 goto KLO  if $tmp  is  zero;
$tmp = $ch - 'p';
 goto KLP  if $tmp  is  zero;
$tmp = $ch - 'r';
 goto KLR  if $tmp  is  zero;
$tmp = $ch - 's';
 goto KLS  if $tmp  is  zero;
$tmp = $ch - 't';
 goto KLT  if $tmp  is  zero;
$tmp = $ch - 'u';
 goto KLU  if $tmp  is  zero;
$tmp = $ch - 'w';
 goto KLW  if $tmp  is  zero;
$tmp = $ch - 'z';
 goto KLZ  if $tmp  is  zero;
 return $r;
KLA:
$tbl = &KeywordTable;
$tbl = $tbl + 342;
 return  call KeywordLookupRange($tbl, 59, $start, $len);
KLB:
$tbl = &KeywordTable;
$r =  call KeywordLookupRange($tbl, 0, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 27;
$r =  call KeywordLookupRange($tbl, 6, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 138;
$r =  call KeywordLookupRange($tbl, 24, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 383;
$r =  call KeywordLookupRange($tbl, 68, $start, $len);
 return $r;
KLC:
$tbl = &KeywordTable;
$tbl = $tbl + 27;
$r =  call KeywordLookupRange($tbl, 6, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 56;
```

```
$r = call KeywordLookupRange($tbl, 11, $start, $len);
 if $r is nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 104;
$r = call KeywordLookupRange($tbl, 19, $start, $len);
 if $r is nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 138;
$r = call KeywordLookupRange($tbl, 24, $start, $len);
 if $r is nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 234;
$r = call KeywordLookupRange($tbl, 40, $start, $len);
 if $r is nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 403;
$r = call KeywordLookupRange($tbl, 72, $start, $len);
 return $r;
KLD:
$tbl = &KeywordTable;
$r = call KeywordLookupRange($tbl, 0, $start, $len);
 if $r is nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 27;
$r = call KeywordLookupRange($tbl, 6, $start, $len);
 if $r is nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 73;
$r = call KeywordLookupRange($tbl, 14, $start, $len);
 if $r is nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 326;
$r = call KeywordLookupRange($tbl, 56, $start, $len);
 return $r;
KLE:
$tbl = &KeywordTable;
$r = call KeywordLookupRange($tbl, 0, $start, $len);
 if $r is nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 104;
$r = call KeywordLookupRange($tbl, 19, $start, $len);
 if $r is nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 299;
```

```
$r =  call KeywordLookupRange($tbl, 52, $start, $len);
 return $r;
KLG:
$tbl = &KeywordTable;
$tbl = $tbl + 56;
$r =  call KeywordLookupRange($tbl, 11, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 73;
$r =  call KeywordLookupRange($tbl, 14, $start, $len);
 return $r;
KLI:
$tbl = &KeywordTable;
$r =  call KeywordLookupRange($tbl, 0, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 279;
$r =  call KeywordLookupRange($tbl, 47, $start, $len);
 return $r;
KLL:
$tbl = &KeywordTable;
$tbl = $tbl + 27;
$r =  call KeywordLookupRange($tbl, 6, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 73;
$r =  call KeywordLookupRange($tbl, 14, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 104;
$r =  call KeywordLookupRange($tbl, 19, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 234;
$r =  call KeywordLookupRange($tbl, 40, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 263;
$r =  call KeywordLookupRange($tbl, 45, $start, $len);
 return $r;
KLM:
$tbl = &KeywordTable;
$tbl = $tbl + 365;
 return  call KeywordLookupRange($tbl, 64, $start, $len);
KLN:
$tbl = &KeywordTable;
$tbl = $tbl + 279;
$r =  call KeywordLookupRange($tbl, 47, $start, $len);
 if $r  is  nonzero:
   return $r;
```

```
 end
$tbl = &KeywordTable;
$tbl = $tbl + 299;
$r =  call KeywordLookupRange($tbl, 52, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 326;
$r =  call KeywordLookupRange($tbl, 56, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 342;
$r =  call KeywordLookupRange($tbl, 59, $start, $len);
 return $r;
KLO:
$tbl = &KeywordTable;
$tbl = $tbl + 279;
$r =  call KeywordLookupRange($tbl, 47, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 342;
$r =  call KeywordLookupRange($tbl, 59, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 383;
$r =  call KeywordLookupRange($tbl, 68, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 138;
$r =  call KeywordLookupRange($tbl, 24, $start, $len);
 return $r;
KLP:
$tbl = &KeywordTable;
$tbl = $tbl + 104;
$r =  call KeywordLookupRange($tbl, 19, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 138;
$r =  call KeywordLookupRange($tbl, 24, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 198;
$r =  call KeywordLookupRange($tbl, 34, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 279;
$r =  call KeywordLookupRange($tbl, 47, $start, $len);
```

```
 return $r;
KLR:
$tbl = &KeywordTable;
$tbl = $tbl + 56;
$r =  call KeywordLookupRange($tbl, 11, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 73;
$r =  call KeywordLookupRange($tbl, 14, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 177;
$r =  call KeywordLookupRange($tbl, 31, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 234;
$r =  call KeywordLookupRange($tbl, 40, $start, $len);
 return $r;
KLS:
$tbl = &KeywordTable;
$tbl = $tbl + 104;
$r =  call KeywordLookupRange($tbl, 19, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 138;
$r =  call KeywordLookupRange($tbl, 24, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 177;
$r =  call KeywordLookupRange($tbl, 31, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 198;
$r =  call KeywordLookupRange($tbl, 34, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 365;
$r =  call KeywordLookupRange($tbl, 64, $start, $len);
 return $r;
KLT:
$tbl = &KeywordTable;
$tbl = $tbl + 138;
$r =  call KeywordLookupRange($tbl, 24, $start, $len);
 if $r  is  nonzero:
   return $r;
 end
$tbl = &KeywordTable;
$tbl = $tbl + 234;
```

```
  $r =  call KeywordLookupRange($tbl, 40, $start, $len);
   if $r  is  nonzero:
     return $r;
   end
  $tbl = &KeywordTable;
  $tbl = $tbl + 383;
  $r =  call KeywordLookupRange($tbl, 68, $start, $len);
   return $r;
  KLU:
  $tbl = &KeywordTable;
  $tbl = $tbl + 263;
  $r =  call KeywordLookupRange($tbl, 45, $start, $len);
   if $r  is  nonzero:
     return $r;
   end
  $tbl = &KeywordTable;
  $tbl = $tbl + 326;
  $r =  call KeywordLookupRange($tbl, 56, $start, $len);
   if $r  is  nonzero:
     return $r;
   end
  $tbl = &KeywordTable;
  $tbl = $tbl + 177;
  $r =  call KeywordLookupRange($tbl, 31, $start, $len);
   return $r;
  KLW:
  $tbl = &KeywordTable;
  $tbl = $tbl + 27;
  $r =  call KeywordLookupRange($tbl, 6, $start, $len);
   if $r  is  nonzero:
     return $r;
   end
  $tbl = &KeywordTable;
  $tbl = $tbl + 138;
  $r =  call KeywordLookupRange($tbl, 24, $start, $len);
   if $r  is  nonzero:
     return $r;
   end
  $tbl = &KeywordTable;
  $tbl = $tbl + 383;
  $r =  call KeywordLookupRange($tbl, 68, $start, $len);
   return $r;
  KLZ:
  $tbl = &KeywordTable;
  $tbl = $tbl + 279;
   return  call KeywordLookupRange($tbl, 47, $start, $len);
end
```

SregLookup walks `SregTable` searching for an entry whose name matches the byte range `[$start..$start+$len)`. On a hit it returns `index + 1` (where `index` is the MMIX special-register number 0–31 stored in the entry); on a miss it returns 0. The `+1` bias lets the caller distinguish `$rB` (index 0) from an unknown suffix with a single `is zero` test.

Each table entry is laid out as a one-byte length, that many name bytes, and a trailing one-byte index. A zero length byte terminates the table. The structure therefore mirrors `KeywordTable` but with an extra byte per entry for the index payload.

⟨ ***Lexer procedures*** ⟩ +≡

```
def SregLookup($start, $len) -> $r:
  register $p, $elen, $tmp, $i, $sp, $kp, $ch1, $ch2;
 $p = &SregTable;
 SRLLoop:


 $elen = [$p].bu;
  if $elen  is  zero:
    return 0;
  end
 $p = $p + 1;
 $tmp = $elen - $len;
  goto SRLNext  if $tmp  is  nonzero;
 $i = 0;
 SRLCmp:
 $tmp = $i - $len;
  goto SRLMatch  if $tmp  is  zero;
 $sp = $start + $i;
 $kp = $p + $i;
 $ch1 = [$sp].bu;
 $ch2 = [$kp].bu;
 $tmp = $ch1 - $ch2;
  goto SRLNext  if $tmp  is  nonzero;
 $i = $i + 1;
  goto SRLCmp;
 SRLNext:
 $p = $p + $elen;
 $p = $p + 1;                    # skip index byte of this entry
  goto SRLLoop;
 SRLMatch:
 $p = $p + $elen;
 $ch1 = [$p].bu;
 $ch1 = $ch1 + 1;                # +1 bias so 0 means miss
  return $ch1;
end
```

`TableLookup` is a generic version of `KeywordLookup` parameterised by the table base address. It searches a length-prefixed, zero-length -terminated table and returns the 1-based index of the first matching entry (or 0 on miss). It is used by the post-identifier classifier to query `UnaryOpTable` and `WydeOpTable`, and by `TLScanDot` to query `SizeTable`. `KeywordLookup` is retained as a separate entry point so that the common keyword-recognition path in `TLScanIdent` does not pay the extra indirection cost of a table argument.

⟨ ***Lexer procedures*** ⟩ +≡

```
def TableLookup($tbl, $start, $len) -> $r:
  register $p, $elen, $tmp, $i, $sp, $kp, $ch1, $ch2, $idx;
 $p = $tbl;
 $idx = 0;


 TL2Loop:
 $elen = [$p].bu;
  if $elen  is  zero:
    return 0;
  end
 $idx = $idx + 1;
 $p = $p + 1;
```

```
    $tmp = $elen - $len;
     goto TL2Next  if $tmp  is  nonzero;
    $i = 0;
    TL2Cmp:
    $tmp = $i - $len;
     if $tmp  is  zero:
       return $idx;
     end
    $sp = $start + $i;
    $kp = $p + $i;
    $ch1 = [$sp].bu;
    $ch2 = [$kp].bu;
    $tmp = $ch1 - $ch2;
     goto TL2Next  if $tmp  is  nonzero;
    $i = $i + 1;
     goto TL2Cmp;
    TL2Next:
    $p = $p + $elen;
     goto TL2Loop;
  end
```

`SregStyleLookup` is the `TableLookup` counterpart for tables whose entries carry a trailing index byte after the name payload (the `SregTable` layout). `DotKwTable` reuses this layout so that its three entries can map directly to the non-contiguous `KW_CMPEDOT` / `KW_EQLEDOT` / `KW_UNEDOT` codes without the caller needing an offset adjustment. The return value is `index_byte + 1` on hit (so zero unambiguously signals miss) and 0 otherwise.

⟨ ***Lexer procedures*** ⟩ +≡

```
  def SregStyleLookup($tbl, $start, $len) -> $r:
    register $p, $elen, $tmp, $i, $sp, $kp, $ch1, $ch2;
   $p = $tbl;
   SSLLoop:


   $elen = [$p].bu;
    if $elen  is  zero:
      return 0;
    end
   $p = $p + 1;
   $tmp = $elen - $len;
    goto SSLNext  if $tmp  is  nonzero;
   $i = 0;
   SSLCmp:
   $tmp = $i - $len;
    goto SSLMatch  if $tmp  is  zero;
   $sp = $start + $i;
   $kp = $p + $i;
   $ch1 = [$sp].bu;
   $ch2 = [$kp].bu;
   $tmp = $ch1 - $ch2;
    goto SSLNext  if $tmp  is  nonzero;
   $i = $i + 1;
    goto SSLCmp;
   SSLNext:
   $p = $p + $elen;
   $p = $p + 1;                    # skip index byte
    goto SSLLoop;
   SSLMatch:
   $p = $p + $elen;
```

```
    $ch1 = [$p].bu;
    $ch1 = $ch1 + 1;              # +1 bias so 0 means miss
     return $ch1;
end
```

18. Parser: cursor and node allocation.

This section establishes the two primitive layers every parser routine builds on: a token cursor for reading `TokenBuf` one token at a time with lookahead, and a node allocator for appending AST nodes to `NodeBuf`. Neither layer knows anything about grammar; the grammar rules themselves are the subject of the later parser sections.

The cursor is a single global `$cur_tok`, a zero-based token index into `TokenBuf`. Every primitive is expressed as a procedure so that the choice of representation (packed token records, byte offsets vs. record indices) can be changed in one place. Lookahead never "unreads" tokens; routines instead peek with an offset and advance only when they commit.

AST nodes are 64-byte records laid out as declared in the constants section (`ND_KIND`, `ND_LINE`, `ND_COL`, `ND_A`, `ND_B`, `ND_C`, `ND_D`, `ND_NEXT`). `NewNode` returns the byte offset of the allocated record, which doubles as the "node handle" stored in `ND_NEXT` chains and in other nodes' payload fields. Handle 0 is reserved as the end-of-list sentinel, so the very first allocation must skip past byte 0; `$node_end` is primed to `NODE_HEADER_SIZE` at program start for exactly this reason.

The parser's global state is a single register, `$cur_tok`, plus `$parse_err_ct` (a count used later by the parser driver to decide whether to proceed to code generation after a recoverable error). Both are declared as `greg`s so that every parser procedure can read and write them without passing a cursor argument through the grammar.

⟨ ***Data segment*** ⟩ +≡

```
greg $cur_tok       = 0;     # index of next unread token in TokenBuf
greg $parse_err_ct  = 0;     # number of parse errors seen so far
greg $ast_root      = 0;     # handle of the NK_PROGRAM root node after ParseProgram
```

Synthetic per-call inline-string-literal state. These would be `greg`s, but the global-register budget is already saturated, so we stash both fields on the reserved sentinel node (handle `0`), which is never linked into any `ND_NEXT` chain and whose other fields are dead. `ND_A` holds the chain head; `ND_B` holds the unique-name counter. The return-expression implementation also parks its current-procedure output-list state here: `ND_C` holds the current procedure's out-parameter list head, and `ND_D` holds the starting slot index for auto-assigned out-parameters. Helper accessors keep these hacks contained.

⟨ ***Parser procedures*** ⟩ ≡

```
def InlineStrChainGet() -> $r:
  return  call NodeGet(0, ND_A);
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def InlineStrChainSet($v):
  call NodeSet(0, ND_A, $v);
  return;
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def InlineStrCtrNext() -> $r:
  register $next;
 $r =  call NodeGet(0, ND_B);
 $next = $r + 1;
  call NodeSet(0, ND_B, $next);
  return $r;
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def CurProcOutsGet() -> $r:
  return  call NodeGet(0, ND_C);
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def CurProcOutsSet($v):
  call NodeSet(0, ND_C, $v);
  return;
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def CurOutStartSlotGet() -> $r:
  return  call NodeGet(0, ND_D);
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def CurOutStartSlotSet($v):
  call NodeSet(0, ND_D, $v);
  return;
end
```

⟨ ***Data segment*** ⟩ +≡

```
greg $label_seq     = 0;      # monotonic counter for generator-synthesised labels
greg $BreakLbl      = 0;      # numeric ID of the innermost enclosing loop's break target (0 =
 none)
greg $ContLbl       = 0;      # numeric ID of the innermost enclosing loop's continue target (0
 = none)
greg $main_emitted = 0;       # set once |Main IS @| has been written to |OutputBuf|
greg $user_loc_seen = 0;      # set once an explicit |loc| directive has been emitted
 (raw/kernel mode)
greg $gen_in_proc   = 0;      # generator scope flag: 0 at top level, 1 inside a def body
greg $last_callbase = 19;    # call-window base used by the most recent GenCall PUSHJ/PUSHGO
greg $GlobalAliasBuf[ALIAS_MAX * ALIAS_ENTRY_SIZE];
greg $LocalAliasBuf[ALIAS_MAX * ALIAS_ENTRY_SIZE];
greg $GlobalAliasCnt = 0;
greg $LocalAliasCnt  = 0;
greg $SymBuf[SYM_MAX * SYM_ENTRY_SIZE];
greg $SymCnt = 0;
greg $SymHashTbl[SYM_HASH_SIZE * 8];
greg $ProcNameOff = 0;
greg $ProcNameLen = 0;
greg $ProcLabelBuf[PROC_LABEL_MAX * PROC_LABEL_ENTRY_SIZE];
greg $ProcLabelCnt = 0;
```

The main program must also advance `$node_end` past node 0 before the parser runs; otherwise the first `NewNode` would return handle 0 and collide with the end-of-list sentinel. This one-line init is done in the top-level `@<Program@>` chunk right before the call to `ParseProgram`.

⟨ ***Program*** ⟩ +≡

```
$node_end = NODE_HEADER_SIZE;
```

Token-field accessors. Each takes a token index and returns the requested field. `TokLine` and `TokCol` unpack the packed line/column octa that `EmitToken` stores in `TK_LINE` (line in the low 32 bits, column in the high 32 bits).

⟨ ***Parser procedures*** ⟩ ≡

```
def TokAddr($idx) -> $r:
  register $tmp;
 $tmp = $idx << 5;                    # * TOKEN_ENTRY_SIZE (= 32)
 $r = $tmp + $TokenBuf;
  return $r;
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def TokKind($idx) -> $r:
  register $addr;
 $addr = $idx << 5;                    # * TOKEN_ENTRY_SIZE
 $addr = $addr + $TokenBuf;
 $r = [$addr + TK_KIND].o;
  return $r;
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def TokValue($idx) -> $r:
  register $addr;
 $addr = $idx << 5;
 $addr = $addr + $TokenBuf;
 $r = [$addr + TK_VALUE].o;
  return $r;
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def TokLen($idx) -> $r:
  register $addr;
 $addr = $idx << 5;
 $addr = $addr + $TokenBuf;
 $r = [$addr + TK_LEN].o;
  return $r;
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def TokLine($idx) -> $r:
  register $addr, $packed, $mask;
 $addr = $idx << 5;
 $addr = $addr + $TokenBuf;
 $packed = [$addr + TK_LINE].o;
 $mask = #100000000;
 $mask = $mask - 1;
 $r = $packed & $mask;
  return $r;
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def TokCol($idx) -> $r:
  register $addr, $packed;
 $addr = $idx << 5;
 $addr = $addr + $TokenBuf;
 $packed = [$addr + TK_LINE].o;
 $r = $packed >>u 32;
  return $r;
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def ErrTokHalt($msg, $idx):
  register $line, $col;
 $line =  call TokLine($idx);
 $col =  call TokCol($idx);
  call ErrHaltAt($msg, $line, $col);
  return;
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def ErrCurTokHalt($msg):
  call ErrTokHalt($msg, $cur_tok);
  return;
end
```

Cursor primitives. `Peek` returns the kind of the token at the current cursor without advancing; `PeekAt` does the same with an offset (`PeekAt(0)` is equivalent to `Peek`; `PeekAt(1)` is the single-token lookahead used by expression parsing). `Advance` unconditionally moves the cursor forward by one token. Callers must ensure they never advance past `TK_EOF`; `Tokenize` always emits a trailing `TK_EOF`, so a parser loop that halts on `TK_EOF` before advancing is safe.

⟨ ***Parser procedures*** ⟩ ≡

```
def Peek() -> $r:
  return  call TokKind($cur_tok);
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def PeekAt($off) -> $r:
  register $idx;
 $idx = $cur_tok + $off;
  return  call TokKind($idx);
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def Advance():
 $cur_tok = $cur_tok + 1;
  return;
end
```

`Expect` consumes a token of a given kind or halts with a parse error. `ExpectKw` does the same for keyword tokens, checking both `TK_KW` and the keyword code in `TK_VALUE`. A proper parser will eventually want a recoverable-error path here (report, resync, keep going), but for the bootstrap a fatal halt is adequate: the compiler is a batch tool and the user can re-run after fixing the source.

`MatchKind` and `MatchKw` are non-consuming predicates used for grammar dispatch. `MatchKind` returns 1 if the current token has the given kind and 0 otherwise. `MatchKw` additionally requires the token to be `TK_KW` with the given keyword code. Neither advances the cursor, so a typical parser routine does `if MatchKw(KW_IF): ...` and then `ExpectKw(KW_IF)` (or `Advance`) once it has committed to the branch.

⟨ ***Parser procedures*** ⟩ ≡

```
def AtEof() -> $r:
  return  call MatchKind(TK_EOF);
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def MatchKind($kind) -> $r:
  register $k, $tmp;
 $k =  call Peek();
 $tmp = $k - $kind;
  if $tmp  is  zero:
    return 1;
  end
  return 0;
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def MatchKw($code) -> $r:
  register $k, $v, $tmp;
 $k =  call Peek();
 $tmp = $k - TK_KW;
  if $tmp  is  nonzero:
    return 0;
  end
 $v =  call TokValue($cur_tok);
 $tmp = $v - $code;
  if $tmp  is  zero:
    return 1;
  end
  return 0;
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def Expect($kind):
  register $k, $tmp;
 $k =  call Peek();
 $tmp = $k - $kind;
  if $tmp  is  nonzero:
    call ErrCurTokHalt("plmmix-compiler: parse error: unexpected token kind\n");
  end
  call Advance();
  return;
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def ExpectKw($code):
  register $k, $v, $tmp;
 $k =  call Peek();
 $tmp = $k - TK_KW;
  if $tmp  is  nonzero:
    call ErrCurTokHalt(&ErrParseKw);
  end
 $v =  call TokValue($cur_tok);
 $tmp = $v - $code;
  if $tmp  is  nonzero:
    call ErrCurTokHalt(&ErrParseKw);
  end
  call Advance();
  return;
end
```

Node allocation. `NewNode` reserves a `NODE_HEADER_SIZE`-byte record at the current end of `NodeBuf`, writes `$kind`, line, and column into the header (taking position from the current token), zero-fills `ND_A` through `ND_NEXT`, and returns the byte offset of the new record as its handle. Arena overflow halts via `ErrNodeFull`.

Callers that allocate a node larger than the 64-byte header (for example a procedure-declaration node with an extra trailing octa for the parameter count) must call `NewNodeExtra` instead. That variant accepts the number of additional octas to reserve and zero-fill; all extra octas live in the region |[handle + 64 .. handle + 64 + 8 * extra)|.

⟨ ***Parser procedures*** ⟩ ≡

```
def NewNode($kind) -> $handle:
  return  call NewNodeExtra($kind, 0);
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def NewNodeExtra($kind, $extra_octas) -> $handle:
  register $size, $base, $tmp, $limit, $off;
 $size = $extra_octas << 3;          # * 8
 $size = $size + NODE_HEADER_SIZE;


 $limit = $node_end + $size;
 $tmp = $limit  cmpu NODE_POOL_MAX;
  if $tmp  is  pos:
    call ErrHalt("plmmix-compiler: AST arena full\n");
  end
 $handle = $node_end;
 $base = $NodeBuf + $handle;
 [$base + ND_KIND].o = $kind;
 $tmp =  call TokLine($cur_tok);
 [$base + ND_LINE].o = $tmp;
 $tmp =  call TokCol($cur_tok);
 [$base + ND_COL].o = $tmp;
 [$base + ND_A].o    = 0;
 [$base + ND_B].o    = 0;
 [$base + ND_C].o    = 0;
 [$base + ND_D].o    = 0;
 [$base + ND_NEXT].o = 0;
  # Zero-fill extra trailing octas.
 $off = 0;
NNEClear:
 $tmp = $off - $size;
  goto NNEDone  if $tmp  is  nonneg;
 $tmp = $off + NODE_HEADER_SIZE;
 $tmp = $tmp - 8;
 $tmp = $base + $tmp;
 [$tmp].o = 0;
 $off = $off + 8;
  goto NNEClear;
NNEDone:
 $node_end = $limit;
  return $handle;
end
```

Node field getters and setters. These wrap the obvious pointer arithmetic so that parser code reads like structured field access even though the underlying layout is a flat byte arena. The `$field` argument is one of `ND_KIND` through `ND_NEXT`, or a `NODE_HEADER_SIZE + 8 * k` offset for extra trailing octas.

⟨ ***Parser procedures*** ⟩ ≡

```
def NodeSet($handle, $field, $val):
  register $addr;
 $addr = $NodeBuf + $handle;
 [$addr + $field].o = $val;
  return;
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def NodeGet($handle, $field) -> $r:
  register $addr;
 $addr = $NodeBuf + $handle;
 $r = [$addr + $field].o;
  return $r;
end
```

⟨ ***Parser procedures*** ⟩ ≡

```
def ErrNodeHalt($msg, $node):
  register $line, $col;
 $line =  call NodeGet($node, ND_LINE);
 $col =  call NodeGet($node, ND_COL);
  call ErrHaltAt($msg, $line, $col);
  return;
end
```

`NodeAppend` links `$node` onto the end of an `ND_NEXT` chain rooted at `$head`. If `$head` is 0 the chain was empty and the new head is simply `$node`; otherwise we walk to the tail and patch its `ND_NEXT`. Returns the (possibly new) head so the caller can keep its root pointer up to date with a single assignment.

The walk is O(n) per append, which is fine for the short lists the parser builds (top-level decls, statements in a body, params in a def). If any list ever grows large enough to matter, callers can switch to maintaining an explicit tail pointer alongside the head.

⟨ ***Parser procedures*** ⟩ ≡

```
def NodeAppend($head, $node) -> $newhead:
  register $cur, $next;
  goto NAEmpty  if $head  is  zero;
 $cur = $head;
 NALoop:
 $next =  call NodeGet($cur, ND_NEXT);
  goto NASet  if $next  is  zero;
 $cur = $next;
  goto NALoop;
 NASet:
  call NodeSet($cur, ND_NEXT, $node);
  return $head;
 NAEmpty:
  return $node;
end
```

Parser error messages. The later parser sections add more specific ones; the two below are used by the `Expect` primitives above and are therefore declared now.

⟨ ***Data segment*** ⟩ +≡

```
byte ErrParseKw   "plmmix-compiler: parse error: expected keyword not found\n", 0;
```

19. Parser: program and simple top-level declarations.

This section installs the real `ParseProgram` driver and the first batch of top-level-declaration parsers—the ones whose syntax does not require a general expression parser. That covers `segment data`, `segment code`, `prefix`, `local`, and `pool`. The declarations that do need expressions (`greg`, `const`, `loc`, `reserve`, and the data directives `byte`/`wyde`/`tetra`/`octa`) and the `def` procedure form are added in a later section.

`ExpectLabelInto` consumes a `TK_LABEL` token and records the interned-string offset and length on a caller-supplied node. Many declaration nodes store a name in `ND_A` (pool offset) with the length in `ND_B`; centralising the pattern here keeps the parsers below compact. `ExpectRegNameInto` is the analogous helper for `TK_REGNAME` (a `$name` register identifier).

⟨ ***Parser procedures*** ⟩ +≡

```
def ExpectLabelInto($handle):
  register $val, $len;
 $val =  call TokValue($cur_tok);
 $len =  call TokLen($cur_tok);
  call Expect(TK_LABEL);
  call NodeSet($handle, ND_A, $val);
  call NodeSet($handle, ND_B, $len);
  return;
end
```

⟨ ***Parser procedures*** ⟩ +≡

```
def ExpectRegNameInto($handle):
  register $val, $len;
 $val =  call TokValue($cur_tok);
 $len =  call TokLen($cur_tok);
  call Expect(TK_REGNAME);
  call NodeSet($handle, ND_A, $val);
  call NodeSet($handle, ND_B, $len);
  return;
end
```

`ParseProgram` allocates the root `NK_PROGRAM` node, then loops over top-level declarations until end-of-file, linking each one onto the ND_A chain rooted at the program node. The root handle is left in the global `$ast_root` for the generator pass to pick up.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseProgram():
  register $root, $head, $item, $cur, $prev, $kind, $tmp, $tail, $next;
  register $chain;
 $root =  call NewNode(NK_PROGRAM);


 $head = 0;
 $tail = 0;
 PPLoop:
 $item =  call AtEof();
  goto PPDone  if $item  is  nonzero;
 $item =  call ParseTopLevel();
  goto PPAddFirst  if $head  is  zero;
  call NodeSet($tail, ND_NEXT, $item);
```

```
    $tail = $item;
     goto PPLoop;
    PPAddFirst:
    $head = $item;
    $tail = $item;
     goto PPLoop;
    PPDone:
    $chain =  call InlineStrChainGet();
     goto PPSetRoot  if $chain  is  zero;
     # Splice synthesised inline-string declarations into the data segment
     # (just before the first |segment code;| directive, or at end if none).
    $cur = $head;
    $prev = 0;
    PPISLoop:
     goto PPISEnd  if $cur  is  zero;
    $kind =  call NodeGet($cur, ND_KIND);
    $tmp = $kind - NK_SEGCODE;
     goto PPISHere  if $tmp  is  zero;
    $tmp = $kind - NK_DEF;
     goto PPISHere  if $tmp  is  zero;
    $prev = $cur;
    $cur =  call NodeGet($cur, ND_NEXT);
     goto PPISLoop;
    PPISHere:
     if $prev  is  zero:
      $head = $chain;
    else:
       call NodeSet($prev, ND_NEXT, $chain);
     end
    $tail = $chain;
    PPISTail:
    $next =  call NodeGet($tail, ND_NEXT);
     goto PPISTailDone  if $next  is  zero;
    $tail = $next;
     goto PPISTail;
    PPISTailDone:
     call NodeSet($tail, ND_NEXT, $cur);
     goto PPClear;
    PPISEnd:
     if $head  is  zero:
      $head = $chain;
    else:
       call NodeSet($tail, ND_NEXT, $chain);
     end
    PPClear:
     call InlineStrChainSet(0);
    PPSetRoot:
     call NodeSet($root, ND_A, $head);
    $ast_root = $root;
     return;
   end
```

`ParseTopLevel` dispatches on the current token's keyword code. When no top-level directive keyword matches, it falls back to `ParseStatement` so that bare top-level statements (including `Label:`) are accepted.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseTopLevel() -> $r:
  register $m, $k, $k2, $tmp;
 $m =  call MatchKw(KW_SEGMENT);
  if $m  is  nonzero:


    return  call ParseSegment();
  end
 $m =  call MatchKw(KW_PREFIX);
  if $m  is  nonzero:
    return  call ParsePrefix();
  end
 $m =  call MatchKw(KW_LOCAL);
  if $m  is  nonzero:
    return  call ParseLocal();
  end
 $m =  call MatchKw(KW_POOL);
  if $m  is  nonzero:
    return  call ParsePool();
  end
 $m =  call MatchKw(KW_CONST);
  if $m  is  nonzero:
    return  call ParseConst();
  end
 $m =  call MatchKw(KW_GREG);
  if $m  is  nonzero:
    return  call ParseGreg();
  end
 $m =  call MatchKw(KW_LOC);
  if $m  is  nonzero:
    return  call ParseLoc();
  end
 $m =  call MatchKw(KW_RESERVE);
  if $m  is  nonzero:
    return  call ParseReserve();
  end
 $m =  call MatchKw(KW_BYTE);
  if $m  is  nonzero:
    return  call ParseDataDir();
  end
 $m =  call MatchKw(KW_WYDE);
  if $m  is  nonzero:
    return  call ParseDataDir();
  end
 $m =  call MatchKw(KW_TETRA);
  if $m  is  nonzero:
    return  call ParseDataDir();
  end
 $m =  call MatchKw(KW_OCTA);
  if $m  is  nonzero:
    return  call ParseDataDir();
  end
 $m =  call MatchKw(KW_DEF);
  if $m  is  nonzero:
    return  call ParseDef();
  end
 $k =  call Peek();
 $tmp = $k - TK_LABEL;
  if $tmp  is  nonzero:
    return  call ParseStatement();
  end
```

```
  $k2 = call PeekAt(1);
  $tmp = $k2 - TK_COLON;
  if $tmp is nonzero:
    return call ParseStatement();
  end
  return call ParseStatement();
end
```

ParseSegment recognises `segment data $Base;` and |segment code;|. The optional base-register name of a data segment is captured into ND_A/ND_B of the NK_SEGDATA node; a missing base leaves both zero, which the generator treats as "use the default data base register". Register-like segment base names must use the `$` sigil, keeping them visually distinct from ordinary labels.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseSegment() -> $r:
  register $m, $tmp;
  call ExpectKw(KW_SEGMENT);
  $m = call MatchKw(KW_DATA);
  goto PSData if $m is nonzero;
  $m = call MatchKw(KW_CODE);
  goto PSCode if $m is nonzero;
  call ErrCurTokHalt("plmmix-compiler: parse error: expected `data' or `code' after
    `segment'\n");
 PSData:
  call ExpectKw(KW_DATA);
  $r = call NewNode(NK_SEGDATA);
  $m = call MatchKind(TK_REGNAME);
  if $m is nonzero:
    call ExpectRegNameInto($r);
  end
 PSDataDone:
  call Expect(TK_SEMI);
  return $r;
 PSCode:
  call ExpectKw(KW_CODE);
  $r = call NewNode(NK_SEGCODE);
  call Expect(TK_SEMI);
  return $r;
end
```

ParsePrefix parses `prefix NAME;` or a bare `prefix;` (which clears any accumulated prefix). The name, if present, is stored on the NK_PREFIX node in the usual pool-offset/length pair.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParsePrefix() -> $r:
  register $m;
  call ExpectKw(KW_PREFIX);
  $r = call NewNode(NK_PREFIX);
  $m = call MatchKind(TK_LABEL);
  if $m is nonzero:
    call ExpectLabelInto($r);
  end
  call Expect(TK_SEMI);
  return $r;
end
```

`ParseLocal` parses `local $N;`—the declaration of a local register. `TK_REGISTER` carries the register number in `TK_VALUE`, so we store that directly in `ND_A` and leave `ND_B` zero.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseLocal() -> $r:
  register $num;
  call ExpectKw(KW_LOCAL);
 $r =  call NewNode(NK_LOCAL);
 $num =  call TokValue($cur_tok);
  call Expect(TK_REGISTER);
  call NodeSet($r, ND_A, $num);
  call Expect(TK_SEMI);
  return $r;
end
```

`ParsePool` parses a bare `pool;` directive, which asks the generator to emit a literal pool at that point in the instruction stream. No operands.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParsePool() -> $r:
  call ExpectKw(KW_POOL);
 $r =  call NewNode(NK_POOL);
  call Expect(TK_SEMI);
  return $r;
end
```

20. Parser: constant expressions and data declarations.

This section builds the constant-expression AST parser that the remaining top-level declarations need, then wires the five forms `const`, `greg`, `loc`, `reserve`, and the data directives `byte`/`wyde`/`tetra`/`octa`.

The expression grammar is the textbook additive/multiplicative cascade with parenthesised subexpressions and unary minus:

$$expr \to term\ (hbox{+} \mid \texttt{-}\ term)^*$$
$$term \to atom\ (hbox{*} \mid \texttt{/}\ atom)^*$$
$$atom \to \texttt{-}\ atom \mid \texttt{(}\ expr\ \texttt{)} \mid integer \mid label.$$

Parser output is a small AST rooted at a single `NK_IMM`, `NK_LABELREF`, `NK_UNOP`, or `NK_BINOP` node. Operator codes live in `ND_C` of a `NK_BINOP` or `NK_UNOP`, taken directly from the corresponding `TK_PLUS`/`TK_MINUS`/`TK_STAR`/`TK_SLASH` kind so that no fresh opcode space is needed. We do not evaluate the expression at parse time—the generator walks the tree and emits MMIXAL source text, which `mmixal` resolves. `const` still records the value as an AST rather than an integer; subsequent passes that need compile-time evaluation walk the tree themselves.

`ParseConstExpr` is the public entry. The two internal helpers `ParseConstAdd` and `ParseConstMul` implement left-to-right precedence climbing.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseConstExpr() -> $r:
  return  call ParseConstAdd();
end
```

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseConstAdd() -> $r:
  register $lhs, $rhs, $k, $tmp, $node;
 $lhs =  call ParseConstMul();
 PCALoop:
 $k =  call Peek();
 $tmp = $k - TK_PLUS;
  goto PCAHit  if $tmp  is  zero;
 $tmp = $k - TK_MINUS;
  goto PCAHit  if $tmp  is  zero;
  return $lhs;
 PCAHit:
  call Advance();
 $rhs =  call ParseConstMul();
 $node =  call NewNode(NK_BINOP);
  call NodeSet($node, ND_A, $lhs);
  call NodeSet($node, ND_B, $rhs);
  call NodeSet($node, ND_C, $k);
 $lhs = $node;
  goto PCALoop;
end
```

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseConstMul() -> $r:
  register $lhs, $rhs, $k, $tmp, $node;
 $lhs =  call ParseConstAtom();
 PCMLoop:
 $k =  call Peek();
 $tmp = $k - TK_STAR;
  goto PCMHit  if $tmp  is  zero;
 $tmp = $k - TK_SLASH;
  goto PCMHit  if $tmp  is  zero;
  return $lhs;
 PCMHit:
  call Advance();
 $rhs =  call ParseConstAtom();
 $node =  call NewNode(NK_BINOP);
  call NodeSet($node, ND_A, $lhs);
  call NodeSet($node, ND_B, $rhs);
  call NodeSet($node, ND_C, $k);
 $lhs = $node;
  goto PCMLoop;
end
```

`ParseConstAtom` handles the four leaf/unary forms. A bare identifier is parked in an `NK_LABELREF` node and left for the generator (or a future constant-folding pass) to resolve against the const/label environment.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseConstAtom() -> $r:
  register $k, $tmp, $val, $len, $node, $inner;
 $k =  call Peek();
 $tmp = $k - TK_MINUS;


  goto PCANeg    if $tmp  is  zero;
 $tmp = $k - TK_LPAREN;
  goto PCAParen  if $tmp  is  zero;
 $tmp = $k - TK_IMMEDIATE;
  goto PCAImm    if $tmp  is  zero;
 $tmp = $k - TK_FLOAT;
  goto PCAFlt    if $tmp  is  zero;
 $tmp = $k - TK_LABEL;
  goto PCALab    if $tmp  is  zero;
  call ErrCurTokHalt(&ErrParseExpr);
 PCANeg:
  call Advance();
 $inner =  call ParseConstAtom();
 $node =  call NewNode(NK_UNOP);
  call NodeSet($node, ND_A, $inner);
  call NodeSet($node, ND_C, TK_MINUS);
  return $node;
 PCAParen:
  call Advance();
 $r =  call ParseConstExpr();
  call Expect(TK_RPAREN);
  return $r;
 PCAImm:
 $val =  call TokValue($cur_tok);
  call Advance();
 $node =  call NewNode(NK_IMM);
  call NodeSet($node, ND_A, $val);
  return $node;
 PCAFlt:
 $val =  call TokValue($cur_tok);
 $len =  call TokLen($cur_tok);
  call Advance();
 $node =  call NewNode(NK_FLOAT);
  call NodeSet($node, ND_A, $val);
  call NodeSet($node, ND_B, $len);
  return $node;
 PCALab:
 $val =  call TokValue($cur_tok);
 $len =  call TokLen($cur_tok);
  call Advance();
 $node =  call NewNode(NK_LABELREF);
  call NodeSet($node, ND_A, $val);
  call NodeSet($node, ND_B, $len);
  return $node;
end
```

`ParseConst` parses `const NAME = EXPR;` and emits `NK_CONST` with the name interned in `ND_A`/`ND_B` and the expression handle in `ND_C`.

⟨***Parser procedures***⟩ +≡

```
def ParseConst() -> $r:
  register $expr;
  call ExpectKw(KW_CONST);
 $r =  call NewNode(NK_CONST);
  call ExpectLabelInto($r);
  call Expect(TK_EQUALS);
 $expr =  call ParseConstExpr();
  call NodeSet($r, ND_C, $expr);
  call Expect(TK_SEMI);
  return $r;
end
```

`ParseGreg` parses the three `greg` forms:

- • `greg $NAME;` — bare declaration.
- • `greg $NAME[SIZE];` — base register with `SIZE` bytes reserved after it.
- • `greg $NAME = EXPR;` — base register with a fixed initial value (integer, label, or label±offset).

The declared identifier must be a `TK_REGNAME` (`$`-prefixed name). The name is stored in `ND_A`/`ND_B`; `ND_C` holds the `= EXPR` initialiser handle (or 0); `ND_D` holds the `[SIZE]` reserve-expression handle (or 0).

⟨***Parser procedures***⟩ +≡

```
def ParseGreg() -> $r:
  register $m, $expr;
  call ExpectKw(KW_GREG);
 $r =  call NewNode(NK_GREG);
  call ExpectRegNameInto($r);
 PGAfterName:
 $m =  call MatchKind(TK_LBRACKET);
  goto PGSize   if $m  is  nonzero;
 $m =  call MatchKind(TK_EQUALS);
  goto PGInit   if $m  is  nonzero;
  goto PGSemi;
 PGSize:
  call Expect(TK_LBRACKET);
 $expr =  call ParseConstExpr();
  call NodeSet($r, ND_D, $expr);
  call Expect(TK_RBRACKET);
  goto PGSemi;
 PGInit:
  call Expect(TK_EQUALS);
 $expr =  call ParseConstExpr();
  call NodeSet($r, ND_C, $expr);
 PGSemi:
  call Expect(TK_SEMI);
  return $r;
end
```

`ParseLoc` parses `loc EXPR;`. The reference grammar accepts only a bare integer literal here, but storing a general constant-expr handle costs nothing and keeps the grammar uniform.

⟨***Parser procedures***⟩ +≡

```
def ParseLoc() -> $r:
  register $expr;
  call ExpectKw(KW_LOC);
 $r =  call NewNode(NK_LOC);
 $expr =  call ParseConstExpr();
  call NodeSet($r, ND_A, $expr);
  call Expect(TK_SEMI);
  return $r;
end
```

`ParseReserve` parses `reserve [LABEL] NBYTES;`. The optional leading label names the reserved block; when absent the reservation is anonymous (just pads the current segment). We detect the label by one-token lookahead: if the current token is `TK_LABEL` and the next token is not an operator that would make it part of an expression, we consume it as the name. For the bootstrap we adopt the rule: if the current token is `TK_LABEL` *and* the following token is `TK_IMMEDIATE` or `TK_LPAREN` or `TK_MINUS`, treat the label as the name; otherwise it is the start of the byte-count expression.

⟨***Parser procedures***⟩ +≡

```
def ParseReserve() -> $r:
  register $k, $k2, $tmp, $m, $expr;
  call ExpectKw(KW_RESERVE);
 $r =  call NewNode(NK_RESERVE);
 $m =  call MatchKind(TK_LABEL);
  goto PRNoName  if $m  is  zero;
 $k2 =  call PeekAt(1);
 $tmp = $k2 - TK_IMMEDIATE;
  goto PRName  if $tmp  is  zero;
 $tmp = $k2 - TK_LPAREN;
  goto PRName  if $tmp  is  zero;
 $tmp = $k2 - TK_MINUS;
  goto PRName  if $tmp  is  zero;
 $tmp = $k2 - TK_LABEL;
  goto PRName  if $tmp  is  zero;
 $tmp = $k2 - TK_FLOAT;
  goto PRName  if $tmp  is  zero;
  goto PRNoName;
PRName:
  call ExpectLabelInto($r);
PRNoName:
 $expr =  call ParseConstExpr();
  call NodeSet($r, ND_C, $expr);
  call Expect(TK_SEMI);
  return $r;
end
```

`ParseDataDir` parses one of `byte`/`wyde`/`tetra`/`octa` directives: a keyword followed by a mandatory label, a comma-separated list of data literals, and a semicolon. The size code is recorded in `ND_C` (`KW_BYTE`/`KW_WYDE`/`KW_TETRA`/`KW_OCTA`), the label in `ND_A`/`ND_B`, and the item-list head in `ND_D`. `ParseDataLiteral` returns one leaf node per literal; strings and labels use `NK_STRING` and `NK_LABELREF`, integers (optionally with a unary-minus prefix) use `NK_IMM` or `NK_UNOP`.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseDataDir() -> $r:
  register $k, $tmp, $head, $tail, $item, $m;
 $k = call Peek();
 $tmp = $k - TK_KW;


 if $tmp is nonzero:
   call ErrCurTokHalt(&ErrParseKw);
 end
 $k = call TokValue($cur_tok);
 call Advance();
 $r = call NewNode(NK_DATADIR);
 call NodeSet($r, ND_C, $k);
 call ExpectLabelInto($r);
 $head = 0;
 $tail = 0;
 $item = call ParseDataLiteral();
 $head = $item;
 $tail = $item;
 PDDLoop:
 $m = call MatchKind(TK_COMMA);
 goto PDDDone if $m is zero;
 call Advance();
 $item = call ParseDataLiteral();
 call NodeSet($tail, ND_NEXT, $item);
 $tail = $item;
 goto PDDLoop;
 PDDDone:
 call NodeSet($r, ND_D, $head);
 call Expect(TK_SEMI);
 return $r;
end
```

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseDataLiteral() -> $r:
  register $k, $tmp, $val, $len, $node, $inner;
 $k = call Peek();
 $tmp = $k - TK_STRING;


 goto PDLStr if $tmp is zero;
 $tmp = $k - TK_IMMEDIATE;
 goto PDLImm if $tmp is zero;
 $tmp = $k - TK_MINUS;
 goto PDLNeg if $tmp is zero;
 $tmp = $k - TK_LABEL;
 goto PDLLab if $tmp is zero;
 call ErrCurTokHalt("plmmix-compiler: parse error: expected data literal (integer, string,
   or label)\n");
 PDLStr:
 $val = call TokValue($cur_tok);
 $len = call TokLen($cur_tok);
 call Advance();
 $node = call NewNode(NK_STRING);
 call NodeSet($node, ND_A, $val);
 call NodeSet($node, ND_B, $len);
 return $node;
 PDLImm:
```

```
 $val =  call TokValue($cur_tok);
  call Advance();
 $node =  call NewNode(NK_IMM);
  call NodeSet($node, ND_A, $val);
  return $node;
 PDLNeg:
  call Advance();
 $inner =  call ParseDataLiteral();
 $node =  call NewNode(NK_UNOP);
  call NodeSet($node, ND_A, $inner);
  call NodeSet($node, ND_C, TK_MINUS);
  return $node;
 PDLLab:
 $val =  call TokValue($cur_tok);
 $len =  call TokLen($cur_tok);
  call Advance();
 $node =  call NewNode(NK_LABELREF);
  call NodeSet($node, ND_A, $val);
  call NodeSet($node, ND_B, $len);
  return $node;
end
```

Parser error messages introduced in this section.

⟨***Data segment***⟩ +≡

```
byte ErrParseExpr     "plmmix-compiler: parse error: expected constant expression\n", 0;
```

21. Parser: procedure declarations.

A PL/MMIX procedure is written

extttdef Name(*params*) `[->` *outs*`]` `:` *body* `end`

where `params` and `outs` are comma-separated parameter descriptors and `body` is a list of statements. Each parameter descriptor has one of the forms

- • `$N` — bare register number, no alias.
- • `$name=$N` — alias `$name` for register N.
- • `$name` — named register (position-assigned).

Position-assigned params take successive register numbers starting from 0 within the procedure scope; the actual assignment is the generator's job, so the parser simply records each param's name (if any) and its explicit register (if any).

The `NK_DEF` node carries one extra trailing octa beyond the standard header to store the output-parameter list head. The layout is:

- • `ND_A` = procedure-name pool offset.
- • `ND_B` = procedure-name length.
- • `ND_C` = head of in-parameter list (`NK_PARAM`).
- • `ND_D` = head of body-statement list.
- • extra[0] = head of out-parameter list.

Each NK_PARAM uses ND_A/ND_B for the alias name (zero if none), ND_C for the explicit register number (if any), and ND_D as a flag: 1 if an explicit =$N or bare $N was given, 0 if the register should be auto-assigned by position.

ParseDef parses the procedure header, the input and output parameter lists, and then a full statement body. The body loop calls ParseStatement until the closing end keyword, so the AST shape for procedures now matches the generator's expectations: ND_C holds the input parameter list, extra[0] holds the output list, and ND_D holds the statement list.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseDef() -> $r:
  register $params, $ptail, $outs, $otail, $body, $btail, $item, $m;
  call ExpectKw(KW_DEF);
 $r =  call NewNodeExtra(NK_DEF, 1);


  call ExpectLabelInto($r);
  call Expect(TK_LPAREN);
 $params = 0;
 $ptail = 0;
 $m =  call MatchKind(TK_RPAREN);
  goto PDEndIn  if $m  is  nonzero;
 $item =  call ParseParam();
 $params = $item;
 $ptail = $item;
PDInLoop:
 $m =  call MatchKind(TK_COMMA);
  goto PDEndIn  if $m  is  zero;
  call Advance();
 $item =  call ParseParam();
  call NodeSet($ptail, ND_NEXT, $item);
 $ptail = $item;
  goto PDInLoop;
PDEndIn:
  call Expect(TK_RPAREN);
  call NodeSet($r, ND_C, $params);
 $outs = 0;
 $otail = 0;
 $m =  call MatchKind(TK_ARROW);
  goto PDNoOuts  if $m  is  zero;
  call Advance();
 $item =  call ParseParam();
 $outs = $item;
 $otail = $item;
PDOutLoop:
 $m =  call MatchKind(TK_COMMA);
  goto PDNoOuts  if $m  is  zero;
  call Advance();
 $item =  call ParseParam();
  call NodeSet($otail, ND_NEXT, $item);
 $otail = $item;
  goto PDOutLoop;
PDNoOuts:
  # extra[0] sits at byte offset NODE_HEADER_SIZE (=64) within the node.
  call NodeSet($r, NODE_HEADER_SIZE, $outs);
  call Expect(TK_COLON);
 $body = 0;
 $btail = 0;
PDBodyLoop:
 $m =  call MatchKw(KW_END);
```

```
    goto PDBodyEnd  if $m  is  nonzero;
   $item =  call ParseStatement();
    goto PDBodyFirst  if $body  is  zero;
    call NodeSet($btail, ND_NEXT, $item);
   $btail = $item;
    goto PDBodyLoop;
  PDBodyFirst:
   $body = $item;
   $btail = $item;
    goto PDBodyLoop;
  PDBodyEnd:
    call ExpectKw(KW_END);
    call NodeSet($r, ND_D, $body);
    return $r;
  end
```

`ParseParam` recognises any of the three parameter forms. The first token determines which. A `TK_REGNAME` head is either an alias with an explicit `=$N` or a position-assigned name; a `TK_REGISTER` head is a bare register number with no alias.

⟨ ***Parser procedures*** ⟩ +≡

```
  def ParseParam() -> $r:
    register $k, $tmp, $m, $num;
   $k =  call Peek();
   $r =  call NewNode(NK_PARAM);


   $tmp = $k - TK_REGISTER;
    goto PPNum   if $tmp  is  zero;
   $tmp = $k - TK_REGNAME;
    goto PPRnm   if $tmp  is  zero;
    call ErrCurTokHalt("plmmix-compiler: parse error: expected parameter\n");
  PPNum:
   $num =  call TokValue($cur_tok);
    call Advance();
    call NodeSet($r, ND_C, $num);
    call NodeSet($r, ND_D, 1);
    return $r;
  PPRnm:
    call ExpectRegNameInto($r);
   $m =  call MatchKind(TK_EQUALS);
    goto PPNoEq  if $m  is  zero;
    call Advance();
   $num =  call TokValue($cur_tok);
    call Expect(TK_REGISTER);
    call NodeSet($r, ND_C, $num);
    call NodeSet($r, ND_D, 1);
    return $r;
  PPNoEq:
    call NodeSet($r, ND_D, 0);
    return $r;
  end
```

`ParseStatement` dispatches on the current token to recognise one statement. The handled repertoire now includes declarations, labels, assignments and stores, calls, returns, traps, save/unsave, rounding, wyde-immediate updates, and the structured-control forms `if`/`while`/`loop`/`do`/`begin`, along with `goto`, `break`, and `continue`. Anything still outside the implemented subset halts

with `ErrParseStmtStub` so parser failures remain explicit.

Procedure bodies also allow local `const` declarations. We reuse `ParseConst` so the syntax and AST shape stay identical to top-level constants, and the generator can reuse `GenConst` for emission.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseStatement() -> $r:
  register $k, $k2, $tmp, $m;
 $m = call MatchKw(KW_CONST);
 if $m is nonzero:


   return call ParseConst();
 end
 $m = call MatchKw(KW_REGISTER);
 if $m is nonzero:
   return call ParseRegDecl(KW_REGISTER, NK_REGDECL);
 end
 $m = call MatchKw(KW_BREAK);
 goto PSBrk   if $m is nonzero;
 $m = call MatchKw(KW_CONTINUE);
 goto PSCont  if $m is nonzero;
 $m = call MatchKw(KW_RETURN);
 if $m is nonzero:
   return call ParseReturn();
 end
 $m = call MatchKw(KW_GOTO);
 if $m is nonzero:
   return call ParseGoto();
 end
 $m = call MatchKw(KW_TRAP);
 if $m is nonzero:
   return call ParseTrap();
 end
 $m = call MatchKw(KW_PRELD);
 goto PSCache  if $m is nonzero;
 $m = call MatchKw(KW_PREGO);
 goto PSCache  if $m is nonzero;
 $m = call MatchKw(KW_PREST);
 goto PSCache  if $m is nonzero;
 $m = call MatchKw(KW_SYNCD);
 goto PSCache  if $m is nonzero;
 $m = call MatchKw(KW_SYNCID);
 goto PSCache  if $m is nonzero;
 $m = call MatchKw(KW_SYNC);
 if $m is nonzero:
   return call ParseSync();
 end
 $m = call MatchKw(KW_LDVTS);
 if $m is nonzero:
   return call ParseLdvts();
 end
 $m = call MatchKw(KW_TRIP);
 if $m is nonzero:
   return call ParseTrip();
 end
 $m = call MatchKw(KW_RESUME);
 if $m is nonzero:
   return call ParseResume();
 end
```

```
$m = call MatchKw(KW_SAVE);
 if $m is nonzero:
   return call ParseSaveLike(KW_SAVE, NK_SAVE);
 end
$m = call MatchKw(KW_UNSAVE);
 if $m is nonzero:
   return call ParseSaveLike(KW_UNSAVE, NK_UNSAVE);
 end
$m = call MatchKw(KW_BEGIN);
 if $m is nonzero:
   return call ParseBegin();
 end
$m = call MatchKw(KW_ROUNDING);
 if $m is nonzero:
   return call ParseRounding();
 end
$m = call MatchKw(KW_WHILE);
 if $m is nonzero:
   return call ParseWhile();
 end
$m = call MatchKw(KW_LOOP);
 if $m is nonzero:
   return call ParseLoop();
 end
$m = call MatchKw(KW_DO);
 if $m is nonzero:
   return call ParseDo();
 end
$m = call MatchKw(KW_IF);
 if $m is nonzero:
   return call ParseIf();
 end
$m = call MatchKw(KW_CALL);
 if $m is nonzero:
   return call ParseCallVoid();
 end
$k = call Peek();
$tmp = $k - TK_SREG;
 if $tmp is zero:
   return call ParseAssign();
 end
$tmp = $k - TK_LBRACKET;
 if $tmp is zero:
   return call ParseStore();
 end
 # For a register or regname, peek the next token to disambiguate
 # store-shorthand (`$base[idx].sz = rhs;') from normal assignment.
$tmp = $k - TK_REGISTER;
 goto PSRegL  if $tmp is zero;
$tmp = $k - TK_REGNAME;
 goto PSRegL  if $tmp is zero;
 # A label may introduce a label statement (`L:'), a store-shorthand
 # (`L[off].sz = rhs;'), or nothing valid.
$tmp = $k - TK_LABEL;
 goto PSLabL  if $tmp is zero;
 goto PSStub;
PSRegL:
$k2 = call PeekAt(1);
$tmp = $k2 - TK_LBRACKET;
 if $tmp is zero:
   return call ParseStoreShort();
 end
```

```
    return  call ParseAssign();
  PSLabL:
  $k2 =  call PeekAt(1);
  $tmp = $k2 - TK_COLON;
   goto PSLbl   if $tmp  is  zero;
  $tmp = $k2 - TK_LBRACKET;
   if $tmp  is  zero:
     return  call ParseStoreShort();
   end
   goto PSStub;
  PSStub:
   call ErrCurTokHalt("plmmix-compiler: parse error: statement form not yet implemented\n");
  PSBrk:
   call ExpectKw(KW_BREAK);
  $r =  call NewNode(NK_BREAK);
   call IfCondTail($r);
   call Expect(TK_SEMI);
   return $r;
  PSCont:
   call ExpectKw(KW_CONTINUE);
  $r =  call NewNode(NK_CONTINUE);
   call IfCondTail($r);
   call Expect(TK_SEMI);
   return $r;
  PSCache:
   return  call ParseCacheStmt();
  PSLbl:
  $r =  call NewNode(NK_LABELSTMT);
   call ExpectLabelInto($r);
   call Expect(TK_COLON);
   return $r;
  end
```

`ParseRegDecl` parses `register ...;`. The operand is a comma-separated list of one or more named register aliases. The single-name `=` form uses a `$name` on the left and a register target (`$N` or `$name`) on the right. The list head is stored in `ND_A`.

⟨ ***Parser procedures*** ⟩ +≡

```
  def ParseRegDeclTarget() -> $r:
    register $k, $tmp, $num, $val, $len;
   $k =  call Peek();
   $tmp = $k - TK_REGISTER;
    goto PRDTNum  if $tmp  is  zero;
   $tmp = $k - TK_REGNAME;
    goto PRDTNam  if $tmp  is  zero;
    call ErrCurTokHalt("plmmix-compiler: parse error: expected $N or $name in register
     declaration\n");
   PRDTNum:
   $num =  call TokValue($cur_tok);
    call Advance();
   $r =  call NewNode(NK_REG);
    call NodeSet($r, ND_A, $num);
    return $r;
   PRDTNam:
   $val =  call TokValue($cur_tok);
   $len =  call TokLen($cur_tok);
    call Advance();
   $r =  call NewNode(NK_REGNAME);
    call NodeSet($r, ND_A, $val);
```

```
  call NodeSet($r, ND_B, $len);
  return $r;
end
```

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseRegDecl($kw, $nk) -> $r:
  register $head, $tail, $item, $m, $tmp;
  call ExpectKw($kw);
 $r =  call NewNode($nk);

 $head = 0;
 $tail = 0;
 $item =  call NewNode(NK_REGNAME);
  call ExpectRegNameInto($item);
 $head = $item;
 $tail = $item;
  # Allow the single-name alias form
  #   register $name = $N;
  # and keep the same node shape by storing both refs in ND_A list.
 $tmp = $kw - KW_REGISTER;
  goto PRDLoop  if $tmp  is  nonzero;
 $m =  call MatchKind(TK_EQUALS);
  goto PRDLoop  if $m  is  zero;
  call Advance();
 $item =  call ParseRegDeclTarget();
  call NodeSet($tail, ND_NEXT, $item);
 $tail = $item;
  call NodeSet($r, ND_A, $head);
  call Expect(TK_SEMI);
  return $r;
PRDLoop:
 $m =  call MatchKind(TK_COMMA);
  goto PRDDone  if $m  is  zero;
  call Advance();
 $item =  call NewNode(NK_REGNAME);
  call ExpectRegNameInto($item);
  call NodeSet($tail, ND_NEXT, $item);
 $tail = $item;
  goto PRDLoop;
PRDDone:
  call NodeSet($r, ND_A, $head);
  call Expect(TK_SEMI);
  return $r;
end
```

⟨***Parser procedures***⟩ +≡

```
def ParseRegRef() -> $r:
  register $k, $tmp, $num, $val, $len;
 $k =  call Peek();
 $tmp = $k - TK_REGISTER;


  goto PRRNum  if $tmp  is  zero;
 $tmp = $k - TK_REGNAME;
  goto PRRNam  if $tmp  is  zero;
 $tmp = $k - TK_SREG;
  goto PRRSr   if $tmp  is  zero;
 $tmp = $k - TK_LABEL;
  goto PRRLab  if $tmp  is  zero;
  call ErrCurTokHalt(&ErrParseRegRef);
 PRRNum:
 $num =  call TokValue($cur_tok);
  call Advance();
 $r =  call NewNode(NK_REG);
  call NodeSet($r, ND_A, $num);
  return $r;
 PRRNam:
 $val =  call TokValue($cur_tok);
 $len =  call TokLen($cur_tok);
  call Advance();
 $r =  call NewNode(NK_REGNAME);
  call NodeSet($r, ND_A, $val);
  call NodeSet($r, ND_B, $len);
  return $r;
 PRRSr:
 $num =  call TokValue($cur_tok);
  call Advance();
 $r =  call NewNode(NK_SREG);
  call NodeSet($r, ND_A, $num);
  return $r;
 PRRLab:
 $val =  call TokValue($cur_tok);
 $len =  call TokLen($cur_tok);
 $num =  call SregLookup($val, $len);
 $tmp = $num - 1;
  goto PRRLabErr  if $tmp  is  neg;
  call Advance();
 $r =  call NewNode(NK_SREG);
  call NodeSet($r, ND_A, $tmp);
  return $r;
 PRRLabErr:
  call ErrCurTokHalt(&ErrParseRegRef);
end
```

`ParseGoto` handles `goto LABEL;` and the conditional form `goto LABEL if $r is <polarity>;` where `<polarity>` is one of `zero`/`nonzero`/`neg`/`nonneg`/`pos`/`nonpos`/`odd`/`even`. The destination label sits in `ND_A`/`ND_B` as usual; the optional condition node goes in `ND_C` (zero if unconditional). The relop form (`$a is eq $b`, etc.) shares the condition node shape and will be added to `ParsePolarityCond` later as an alternative tail.

⟨***Parser procedures***⟩ +≡

```
def ParseGoto() -> $r:
  register $m, $cond;
  call ExpectKw(KW_GOTO);
 $r =  call NewNode(NK_GOTO);
 $m =  call MatchKind(TK_LBRACKET);
  goto PGtDir  if $m  is  zero;
  call Advance();
 $cond =  call ParseRegRef();
  call NodeSet($r, ND_A, $cond);
  call NodeSet($r, ND_D, 1);
  call Expect(TK_RBRACKET);
  goto PGtIf;
 PGtDir:
  call ExpectLabelInto($r);
 PGtIf:
 $m =  call MatchKw(KW_IF);
  goto PGtSemi  if $m  is  zero;
  call Advance();
  call ParseBranchHint();
 $cond =  call ParsePolarityCond();
  call NodeSet($r, ND_C, $cond);
 PGtSemi:
  call Expect(TK_SEMI);
  return $r;
end
```

`ParseBranchHint` consumes an optional branch-hint keyword (`likely` or `unlikely`). The current bootstrap generator ignores the hint, but the parser accepts and discards it so surface syntax remains stable even when no branch prediction hint is emitted.

⟨***Parser procedures***⟩ +≡

```
def ParseBranchHint():
  register $m;
 $m =  call MatchKw(KW_LIKELY);
  goto PBHDone  if $m  is  nonzero;
 $m =  call MatchKw(KW_UNLIKELY);
  if $m  is  zero:
    return;
  end
 PBHDone:
  call Advance();
  return;
end
```

`ParseStmtPostfixIf` consumes an optional postfix tail `if [likelyunlikely] cond|` after a statement node and returns either the original node or an `NK_IF` wrapper whose then-body is the single statement. The trailing semicolon is consumed in both cases.

⟨***Parser procedures***⟩ +≡

```
def ParseStmtPostfixIf($stmt) -> $r:
  register $m, $cond, $wrap;
 $m =  call MatchKw(KW_IF);
  goto PSPISemi  if $m  is  zero;
  call Advance();
  call ParseBranchHint();
 $cond =  call ParsePolarityCond();
 $wrap =  call NewNode(NK_IF);
  call NodeSet($wrap, ND_A, $cond);
  call NodeSet($wrap, ND_B, $stmt);
 $stmt = $wrap;
PSPISemi:
  call Expect(TK_SEMI);
  return $stmt;
end
```

`ParsePolarityCond` parses `$r is <polarity>` and returns an `NK_COND` node. Layout: `ND_A` = register-reference child node, `ND_C` = polarity keyword code (`KW_ZERO` through `KW_EVEN`).

⟨***Parser procedures***⟩ +≡

```
def ParsePolarityCond() -> $r:
  register $reg, $pol, $m, $tmp, $k, $rhs;
 $reg =  call ParseRegRef();
 $m =  call IsRelopTok();


  goto PPCRel  if $m  is  nonzero;
  call ExpectKw(KW_IS);
 $m =  call MatchKw(KW_ZERO);
  goto PPCOk  if $m  is  nonzero;
 $m =  call MatchKw(KW_NONZERO);
  goto PPCOk  if $m  is  nonzero;
 $m =  call MatchKw(KW_NEG);
  goto PPCOk  if $m  is  nonzero;
 $m =  call MatchKw(KW_NONNEG);
  goto PPCOk  if $m  is  nonzero;
 $m =  call MatchKw(KW_POS);
  goto PPCOk  if $m  is  nonzero;
 $m =  call MatchKw(KW_NONPOS);
  goto PPCOk  if $m  is  nonzero;
 $m =  call MatchKw(KW_ODD);
  goto PPCOk  if $m  is  nonzero;
 $m =  call MatchKw(KW_EVEN);
  goto PPCOk  if $m  is  nonzero;
  call ErrCurTokHalt("plmmix-compiler: parse error: expected polarity (`zero', `nonzero',
    `neg', ...)\n");
PPCOk:
 $pol =  call TokValue($cur_tok);
  call Advance();
 $r =  call NewNode(NK_COND);
  call NodeSet($r, ND_A, $reg);
  call NodeSet($r, ND_C, $pol);
  return $r;
PPCRel:
 $k =  call Peek();
  call Advance();
 $rhs =  call ParseAtom();
 $r =  call NewNode(NK_COND);
```

```
  call NodeSet($r, ND_A, $reg);
  call NodeSet($r, ND_D, $rhs);
  call NodeSet($r, ND_C, $k);
  return $r;
end
```

`IsRelopTok` returns 1 when the current token is any of the twenty-one comparison tokens (signed, unsigned, and float variants of `==`, `!=`, `<`, `<=`, `>`, `>=`). Used by `ParsePolarityCond` to decide between the `is polarity` tail and the `RELOP rhs` tail. The relop-code stored in `NK_COND.ND_C` is the `TK_` code directly; polarity codes are `KW_` codes (always less than `TK_EQ=13`), so the two code spaces are disjoint and the generator can discriminate by range.

⟨***Parser procedures***⟩ +≡

```
def IsRelopTok() -> $r:
  register $k, $tmp;
 $k =  call Peek();
 $r = 1;


 $tmp = $k - TK_EQ;
  if $tmp  is  zero:
    return $r;
  end
 $tmp = $k - TK_NEQ;
  if $tmp  is  zero:
    return $r;
  end
 $tmp = $k - TK_LT;
  if $tmp  is  zero:
    return $r;
  end
 $tmp = $k - TK_LEQ;
  if $tmp  is  zero:
    return $r;
  end
 $tmp = $k - TK_GT;
  if $tmp  is  zero:
    return $r;
  end
 $tmp = $k - TK_GEQ;
  if $tmp  is  zero:
    return $r;
  end
 $tmp = $k - TK_LTU;
  if $tmp  is  zero:
    return $r;
  end
 $tmp = $k - TK_LEQU;
  if $tmp  is  zero:
    return $r;
  end
 $tmp = $k - TK_GTU;
  if $tmp  is  zero:
    return $r;
  end
 $tmp = $k - TK_GEQU;
  if $tmp  is  zero:
    return $r;
  end
```

```
  $tmp = $k - TK_EQF;
   if $tmp  is  zero:
     return $r;
   end
  $tmp = $k - TK_NEQF;
   if $tmp  is  zero:
     return $r;
   end
  $tmp = $k - TK_LTF;
   if $tmp  is  zero:
     return $r;
   end
  $tmp = $k - TK_LEQF;
   if $tmp  is  zero:
     return $r;
   end
  $tmp = $k - TK_GTF;
   if $tmp  is  zero:
     return $r;
   end
  $tmp = $k - TK_GEQF;
   if $tmp  is  zero:
     return $r;
   end
  $r = 0;
   return $r;
end
```

ParseReturnExpr parses one item in a `return` expression list. Register references (`$N`, `$name`, `$rX`, or a special-register keyword) are parsed via ParseRegRef and produce NK_REG, NK_REGNAME, or NK_SREG nodes. An integer immediate produces an NK_IMM node; a leading `-` is allowed only before an immediate and wraps that immediate in `NK_UNOP/TK_MINUS`. A plain label that does not name a special register produces an NK_LABELREF node, which GenReturn resolves as a compile-time constant alias. A `call` keyword delegates to ParseCallTail and produces an NK_CALL node; the enclosing function must have a `->` clause. When any item in the list is non-register, the enclosing function must declare its return registers via a `->` clause; the expressions are mapped positionally onto those declared registers.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseReturnExpr() -> $r:
  register $k, $tmp, $val;
 $k =  call Peek();
 $tmp = $k - TK_KW;
  if $tmp  is  nonzero:
    return  call ParseRHSExpr();
  end
 $val =  call TokValue($cur_tok);
 $tmp = $val - KW_CALL;
  if $tmp  is  zero:
    return  call ParseCallTail(NK_CALL);
  end
  return  call ParseRHSExpr();
end
```

ParseReturn accepts either a bare `return;`, `return ReturnExpr, ...;`, or the conditional tail `return [...]  if cond;`. Each item in the expression list is parsed by ParseReturnExpr,

which accepts ordinary RHS expressions plus `call ...` when the function declares its return registers via `->`. The list head (or 0) is stored in `ND_A`; the optional condition in `ND_C`.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseReturn() -> $r:
  register $head, $tail, $item, $m;
  call ExpectKw(KW_RETURN);
 $r =  call NewNode(NK_RETURN);


  # A bare `return;' or `return if cond;' has no register list.
 $m =  call MatchKind(TK_SEMI);
  goto PRtDone  if $m  is  nonzero;
 $m =  call MatchKw(KW_IF);
  goto PRtIf    if $m  is  nonzero;
 $head = 0;
 $tail = 0;
 $item =  call ParseReturnExpr();
 $head = $item;
 $tail = $item;
 PRtLoop:
 $m =  call MatchKind(TK_COMMA);
  goto PRtEnd  if $m  is  zero;
  call Advance();
 $item =  call ParseReturnExpr();
  call NodeSet($tail, ND_NEXT, $item);
 $tail = $item;
  goto PRtLoop;
 PRtEnd:
  call NodeSet($r, ND_A, $head);
 PRtIf:
  call IfCondTail($r);
 PRtDone:
  call Expect(TK_SEMI);
  return $r;
end
```

`IfCondTail` consumes an optional `if cond` tail and stores the resulting `NK_COND` node in `ND_C` of its argument. It is shared by `break`, `continue`, and `return` statements.

⟨ ***Parser procedures*** ⟩ +≡

```
def IfCondTail($node):
  register $m, $cond;
 $m =  call MatchKw(KW_IF);
  if $m  is  zero:
    return;
  end
  call Advance();
  call ParseBranchHint();
 $cond =  call ParsePolarityCond();
  call NodeSet($node, ND_C, $cond);
  return;
end
```

`ParseTrap` parses `trap X, Y, Z;`. The three operands are constant expressions—typically small immediates (0..255) or symbolic aliases such as `Halt` or `Fputs`. They are stored as expression-AST

handles in ND_A, ND_B, ND_C.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseTrap() -> $r:
  register $x, $y, $z;
  call ExpectKw(KW_TRAP);
 $r =  call NewNode(NK_TRAP);
 $x =  call ParseConstExpr();
  call Expect(TK_COMMA);
 $y =  call ParseConstExpr();
  call Expect(TK_COMMA);
 $z =  call ParseConstExpr();
  call NodeSet($r, ND_A, $x);
  call NodeSet($r, ND_B, $y);
  call NodeSet($r, ND_C, $z);
  call Expect(TK_SEMI);
  return $r;
end
```

ParseCacheStmt parses the cache-control family `preld/prego/prest/syncd/syncid X, addr;`. The keyword code is stored in ND_A, the immediate/alias expression in ND_B, and the address pair in ND_C/ND_D.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseCacheStmt() -> $r:
  register $kw, $x;
 $kw =  call TokValue($cur_tok);
  call Advance();
 $r =  call NewNode(NK_PRELD);
  call NodeSet($r, ND_A, $kw);
 $x =  call ParseConstExpr();
  call NodeSet($r, ND_B, $x);
  call Expect(TK_COMMA);
  call ParseAddrInto($r, ND_C, ND_D);
  return  call ParseStmtPostfixIf($r);
end
```

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseSync() -> $r:
  register $x;
  call ExpectKw(KW_SYNC);
 $r =  call NewNode(NK_SYNC);
 $x =  call ParseConstExpr();
  call NodeSet($r, ND_A, $x);
  return  call ParseStmtPostfixIf($r);
end
```

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseCswap($dst) -> $r:
  call ExpectKw(KW_CSWAP);
 $r =  call NewNode(NK_PRELD);
  call NodeSet($r, ND_A, KW_CSWAP);
  call NodeSet($r, ND_B, $dst);
  call Expect(TK_LBRACKET);
  call ParseAddrInto($r, ND_C, ND_D);
  call Expect(TK_RBRACKET);
  return  call ParseStmtPostfixIf($r);
end
```

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseLdvts() -> $r:
  register $reg;
  call ExpectKw(KW_LDVTS);
 $r =  call NewNode(NK_PRELD);
  call NodeSet($r, ND_A, KW_LDVTS);
 $reg =  call ParseRegRef();
  call NodeSet($r, ND_B, $reg);
  call Expect(TK_COMMA);
  call ParseAddrInto($r, ND_C, ND_D);
  return  call ParseStmtPostfixIf($r);
end
```

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseTrip() -> $r:
  register $x, $y, $z;
  call ExpectKw(KW_TRIP);
 $r =  call NewNode(NK_TRIP);
 $x =  call ParseConstExpr();
  call Expect(TK_COMMA);
 $y =  call ParseConstExpr();
  call Expect(TK_COMMA);
 $z =  call ParseConstExpr();
  call NodeSet($r, ND_A, $x);
  call NodeSet($r, ND_B, $y);
  call NodeSet($r, ND_C, $z);
  return  call ParseStmtPostfixIf($r);
end
```

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseResume() -> $r:
  register $m, $z;
  call ExpectKw(KW_RESUME);
 $r =  call NewNode(NK_RESUME);
 $m =  call MatchKind(TK_SEMI);
  goto PRsDone  if $m  is  nonzero;
 $m =  call MatchKw(KW_IF);
  goto PRsDone  if $m  is  nonzero;
 $z =  call ParseConstExpr();
  call NodeSet($r, ND_A, $z);
PRsDone:
  return  call ParseStmtPostfixIf($r);
end
```

`ParseSaveLike` factors out `save $r;` and `unsave $r;`, which differ only in keyword and node kind. The register-reference node goes in `ND_A`.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseSaveLike($kw, $nk) -> $r:
  register $reg;
  call ExpectKw($kw);
 $r =  call NewNode($nk);
 $reg =  call ParseRegRef();
  call NodeSet($r, ND_A, $reg);
  call Expect(TK_SEMI);
  return $r;
end
```

`ParseBegin` parses `begin: body end`—a lexical block with no runtime effect beyond scoping aliases. The body-statement list head is stored in `ND_A`. The body loop reuses `ParseStatement`, so nested `begin` blocks and any supported statement form work naturally.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseBegin() -> $r:
  register $head, $tail, $item, $m;
  call ExpectKw(KW_BEGIN);
 $r =  call NewNode(NK_BEGIN);
  call Expect(TK_COLON);
 $head = 0;
 $tail = 0;
PBBLoop:
 $m =  call MatchKw(KW_END);
  goto PBBEnd  if $m  is  nonzero;
 $item =  call ParseStatement();
  goto PBBFirst  if $head  is  zero;
  call NodeSet($tail, ND_NEXT, $item);
 $tail = $item;
  goto PBBLoop;
PBBFirst:
 $head = $item;
 $tail = $item;
  goto PBBLoop;
PBBEnd:
  call ExpectKw(KW_END);
```

```
  call NodeSet($r, ND_A, $head);
  return $r;
end
```

`ParseRounding` parses `rounding MODE;` where `MODE` is one of `nearest`, `zero`, `up`, `down`. The keyword code itself is stored in `ND_A`; the generator maps it to the corresponding ROUND_MODE value when emitting `SETMH` / `PUT rA,...`.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseRounding() -> $r:
  register $m, $tmp, $code;
  call ExpectKw(KW_ROUNDING);
 $r =  call NewNode(NK_ROUNDING);
 $m =  call MatchKw(KW_NEAREST);
  goto PRdOk  if $m  is  nonzero;
 $m =  call MatchKw(KW_ZERO);
  goto PRdOk  if $m  is  nonzero;
 $m =  call MatchKw(KW_UP);
  goto PRdOk  if $m  is  nonzero;
 $m =  call MatchKw(KW_DOWN);
  goto PRdOk  if $m  is  nonzero;
  call ErrCurTokHalt("plmmix-compiler: parse error: expected rounding mode (`nearest',
    `zero', `up', `down')\n");
PRdOk:
 $code =  call TokValue($cur_tok);
  call Advance();
  call NodeSet($r, ND_A, $code);
  call Expect(TK_SEMI);
  return $r;
end
```

`ParseWhile` parses `while cond:  body end`. Layout: `ND_A` = condition node, `ND_B` = body-list head. `ParseLoop` parses `loop:  body end`—an infinite loop terminated by `break` or `return`—with the body head in `ND_A`. Both share a body loop identical in shape to `ParseBegin`.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseWhile() -> $r:
  register $cond, $head, $tail, $item, $m;
  call ExpectKw(KW_WHILE);
 $r =  call NewNode(NK_WHILE);
  call ParseBranchHint();
 $cond =  call ParsePolarityCond();
  call NodeSet($r, ND_A, $cond);
  call Expect(TK_COLON);
 $head = 0;
 $tail = 0;
PWhLoop:
 $m =  call MatchKw(KW_END);
  goto PWhEnd  if $m  is  nonzero;
 $item =  call ParseStatement();
  goto PWhFirst  if $head  is  zero;
  call NodeSet($tail, ND_NEXT, $item);
 $tail = $item;
  goto PWhLoop;
PWhFirst:
 $head = $item;
```

```
    $tail = $item;
     goto PWhLoop;
    PWhEnd:
     call ExpectKw(KW_END);
     call NodeSet($r, ND_B, $head);
     return $r;
    end
```

⟨ ***Parser procedures*** ⟩ +≡

```
    def ParseLoop() -> $r:
      register $head, $tail, $item, $m;
      call ExpectKw(KW_LOOP);
     $r =  call NewNode(NK_LOOP);
      call Expect(TK_COLON);
     $head = 0;
     $tail = 0;
     PLpLoop:
     $m =  call MatchKw(KW_END);
      goto PLpEnd  if $m  is  nonzero;
     $item =  call ParseStatement();
      goto PLpFirst  if $head  is  zero;
      call NodeSet($tail, ND_NEXT, $item);
     $tail = $item;
      goto PLpLoop;
     PLpFirst:
     $head = $item;
     $tail = $item;
      goto PLpLoop;
     PLpEnd:
      call ExpectKw(KW_END);
      call NodeSet($r, ND_A, $head);
      return $r;
    end
```

⟨ ***Parser procedures*** ⟩ +≡

```
    def ParseElif() -> $r:
      register $cond, $head, $tail, $item, $m;
     $r =  call NewNode(NK_ELIF);
      call ParseBranchHint();


     $cond =  call ParsePolarityCond();
      call NodeSet($r, ND_A, $cond);
      call Expect(TK_COLON);
     $head = 0;
     $tail = 0;
     PElLp:
     $m =  call MatchKw(KW_END);
      goto PElDn  if $m  is  nonzero;
     $m =  call MatchKw(KW_ELSE);
      goto PElDn  if $m  is  nonzero;
     $m =  call MatchKw(KW_ELIF);
      goto PElDn  if $m  is  nonzero;
     $item =  call ParseStatement();
      goto PElFirst  if $head  is  zero;
      call NodeSet($tail, ND_NEXT, $item);
```

```
    $tail = $item;
     goto PElLp;
    PElFirst:
    $head = $item;
    $tail = $item;
     goto PElLp;
    PElDn:
     call NodeSet($r, ND_B, $head);
     return $r;
   end
```

⟨ ***Parser procedures*** ⟩ +≡

```
   def ParseDo() -> $r:
     register $head, $tail, $item, $m, $cond;
     call ExpectKw(KW_DO);
    $r =  call NewNode(NK_DO);


     call Expect(TK_COLON);
    $head = 0;
    $tail = 0;
    PDoLoop:
    $m =  call MatchKw(KW_END);
     goto PDoEnd  if $m  is  nonzero;
    $item =  call ParseStatement();
     goto PDoFirst  if $head  is  zero;
     call NodeSet($tail, ND_NEXT, $item);
    $tail = $item;
     goto PDoLoop;
    PDoFirst:
    $head = $item;
    $tail = $item;
     goto PDoLoop;
    PDoEnd:
     call ExpectKw(KW_END);
     call NodeSet($r, ND_A, $head);
    $m =  call MatchKw(KW_WHILE);
     goto PDoRet  if $m  is  zero;
     call ExpectKw(KW_WHILE);
     call ParseBranchHint();
    $cond =  call ParsePolarityCond();
     call NodeSet($r, ND_B, $cond);
     call Expect(TK_SEMI);
    PDoRet:
     return $r;
   end
```

`ParseIf` parses a full `if` statement with an optional `elif` chain and optional `else` body. Layout: `ND_A` = condition, `ND_B` = then-body, `ND_C` = `NK_ELIF` chain, `ND_D` = else-body.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseIf() -> $r:
  register $cond, $head, $tail, $item, $m;
  call ExpectKw(KW_IF);
 $r =  call NewNode(NK_IF);


  call ParseBranchHint();
 $cond =  call ParsePolarityCond();
  call NodeSet($r, ND_A, $cond);
  call Expect(TK_COLON);
 $head = 0;
 $tail = 0;
 PIfLoop:
 $m =  call MatchKw(KW_END);
  goto PIfEnd   if $m  is  nonzero;
 $m =  call MatchKw(KW_ELSE);
  goto PIfElse  if $m  is  nonzero;
 $m =  call MatchKw(KW_ELIF);
  goto PIfElif  if $m  is  nonzero;
 $item =  call ParseStatement();
  goto PIfFirst  if $head  is  zero;
  call NodeSet($tail, ND_NEXT, $item);
 $tail = $item;
  goto PIfLoop;
 PIfFirst:
 $head = $item;
 $tail = $item;
  goto PIfLoop;
 PIfElif:
  call NodeSet($r, ND_B, $head);
 $head = 0;
 $tail = 0;
 PIfElifL:
  call ExpectKw(KW_ELIF);
 $item =  call ParseElif();
  goto PIfElifFirst  if $head  is  zero;
  call NodeSet($tail, ND_NEXT, $item);
 $tail = $item;
  goto PIfElifNext;
 PIfElifFirst:
 $head = $item;
 $tail = $item;
 PIfElifNext:
 $m =  call MatchKw(KW_ELIF);
  goto PIfElifL  if $m  is  nonzero;
  call NodeSet($r, ND_C, $head);
 $head = 0;
 $tail = 0;
 $m =  call MatchKw(KW_ELSE);
  goto PIfElse2  if $m  is  nonzero;
  call ExpectKw(KW_END);
  return $r;
 PIfElse2:
  call ExpectKw(KW_ELSE);
  call Expect(TK_COLON);
 PIfEl2Lp:
 $m =  call MatchKw(KW_END);
  goto PIfEl2Dn  if $m  is  nonzero;
 $item =  call ParseStatement();
  goto PIfEl2First  if $head  is  zero;
  call NodeSet($tail, ND_NEXT, $item);
```

```
    $tail = $item;
     goto PIfEl2Lp;
    PIfEl2First:
    $head = $item;
    $tail = $item;
     goto PIfEl2Lp;
    PIfEl2Dn:
     call ExpectKw(KW_END);
     call NodeSet($r, ND_D, $head);
     return $r;
    PIfElse:
     call NodeSet($r, ND_B, $head);
     call ExpectKw(KW_ELSE);
     call Expect(TK_COLON);
    $head = 0;
    $tail = 0;
    PIfElLp:
    $m =  call MatchKw(KW_END);
     goto PIfElDn  if $m  is  nonzero;
    $item =  call ParseStatement();
     goto PIfElFirst  if $head  is  zero;
     call NodeSet($tail, ND_NEXT, $item);
    $tail = $item;
     goto PIfElLp;
    PIfElFirst:
    $head = $item;
    $tail = $item;
     goto PIfElLp;
    PIfElDn:
     call ExpectKw(KW_END);
     call NodeSet($r, ND_D, $head);
     return $r;
    PIfEnd:
     call ExpectKw(KW_END);
     call NodeSet($r, ND_B, $head);
     return $r;
   end
```

Runtime-expression parser. Unlike the constant-expression parser of the previous parser section, which assumes a tree of purely compile-time values, the runtime form allows any atom that a register-machine instruction can consume—registers, immediates, labels, address-of forms—and permits at most one binary operator joining two atoms. PL/MMIX intentionally forbids multi-step arithmetic on a single line; the user writes explicit temporaries instead.

`ParseAtom` recognises one atom: `$N`, `$name`, `$rX`, an integer (optionally prefixed with unary minus), a label, or `&Label`. Load forms (`[ptr+off].sz`), unary operators, and floating literals are handled here as well.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseAtom() -> $r:
  register $k, $tmp, $val, $len, $node;
 $k =  call Peek();
 $tmp = $k - TK_REGISTER;


  goto PAtReg   if $tmp  is  zero;
 $tmp = $k - TK_REGNAME;
  goto PAtRnm   if $tmp  is  zero;
 $tmp = $k - TK_SREG;
  goto PAtSr    if $tmp  is  zero;
 $tmp = $k - TK_IMMEDIATE;
  goto PAtImm   if $tmp  is  zero;
 $tmp = $k - TK_FLOAT;
  goto PAtFlt   if $tmp  is  zero;
 $tmp = $k - TK_MINUS;
  goto PAtNeg   if $tmp  is  zero;
 $tmp = $k - TK_MINUSU;
  goto PAtNegU  if $tmp  is  zero;
 $tmp = $k - TK_LABEL;
  goto PAtLab   if $tmp  is  zero;
 $tmp = $k - TK_AMP;
  goto PAtAmp   if $tmp  is  zero;
 $tmp = $k - TK_LBRACKET;
  if $tmp  is  zero:
    return  call ParseLoad();
  end
 $tmp = $k - TK_UNARYOP;
  if $tmp  is  zero:
    return  call ParseUnaryOp();
  end
  call ErrCurTokHalt("plmmix-compiler: parse error: expected expression atom\n");
PAtReg:
 $val =  call TokValue($cur_tok);
  call Advance();
 $node =  call NewNode(NK_REG);
  call NodeSet($node, ND_A, $val);
  return  call MaybeIndexedLoad($node);
PAtRnm:
 $val =  call TokValue($cur_tok);
 $len =  call TokLen($cur_tok);
  call Advance();
 $node =  call NewNode(NK_REGNAME);
  call NodeSet($node, ND_A, $val);
  call NodeSet($node, ND_B, $len);
  return  call MaybeIndexedLoad($node);
PAtSr:
 $val =  call TokValue($cur_tok);
  call Advance();
 $node =  call NewNode(NK_SREG);
  call NodeSet($node, ND_A, $val);
  return $node;
PAtImm:
 $val =  call TokValue($cur_tok);
  call Advance();
 $node =  call NewNode(NK_IMM);
  call NodeSet($node, ND_A, $val);
  return $node;
PAtFlt:
 $val =  call TokValue($cur_tok);
 $len =  call TokLen($cur_tok);
```

```
    call Advance();
  $node =  call NewNode(NK_FLOAT);
    call NodeSet($node, ND_A, $val);
    call NodeSet($node, ND_B, $len);
    return $node;
  PAtNeg:
    call Advance();
  $node =  call ParseAtom();
  $r =  call NewNode(NK_UNOP);
    call NodeSet($r, ND_A, $node);
    call NodeSet($r, ND_C, TK_MINUS);
    return $r;
  PAtNegU:
    call Advance();
  $node =  call ParseAtom();
  $r =  call NewNode(NK_UNOP);
    call NodeSet($r, ND_A, $node);
    call NodeSet($r, ND_C, TK_MINUSU);
    return $r;
  PAtLab:
  $val =  call TokValue($cur_tok);
  $len =  call TokLen($cur_tok);
    call Advance();
  $node =  call NewNode(NK_LABELREF);
    call NodeSet($node, ND_A, $val);
    call NodeSet($node, ND_B, $len);
    return  call MaybeIndexedLoad($node);
  PAtAmp:
    call Advance();
  $val =  call TokValue($cur_tok);
  $len =  call TokLen($cur_tok);
    call Expect(TK_LABEL);
  $node =  call NewNode(NK_GETA);
    call NodeSet($node, ND_A, $val);
    call NodeSet($node, ND_B, $len);
    return $node;
  end
```

`IsTokBinop` returns 1 when the current token is one of the binary-operator tokens: arithmetic `+ - * / %` and their unsigned/float variants, bitwise or/xor/tilde operators, shifts, and the scaled adds `2addu`/`4addu`/`8addu`/`16addu`. `IsKwBinop` does the same for keyword-encoded binary operators (`andn`, `orn`, `nand`, `nor`, `nxor`, `mor`, `mxor`, `mux`, `sadd`, `bdif`, `wdif`, `tdif`, `odif`, `cmp`, `cmpu`, `cmpe.`, `eqle.`, `une.`).

⟨ ***Parser procedures*** ⟩ +≡

```
def IsTokBinop($k) -> $r:
  register $tmp;
  $r = 1;
  $tmp = $k - TK_PLUS;

  if $tmp  is  zero:
    return $r;
  end
  $tmp = $k - TK_MINUS;
  if $tmp  is  zero:
    return $r;
  end
  $tmp = $k - TK_STAR;
```

```
 if $tmp is zero:
   return $r;
 end
$tmp = $k - TK_SLASH;
 if $tmp is zero:
   return $r;
 end
$tmp = $k - TK_PERCENT;
 if $tmp is zero:
   return $r;
 end
$tmp = $k - TK_PLUSU;
 if $tmp is zero:
   return $r;
 end
$tmp = $k - TK_MINUSU;
 if $tmp is zero:
   return $r;
 end
$tmp = $k - TK_STARU;
 if $tmp is zero:
   return $r;
 end
$tmp = $k - TK_SLASHU;
 if $tmp is zero:
   return $r;
 end
$tmp = $k - TK_FADD;
 if $tmp is zero:
   return $r;
 end
$tmp = $k - TK_FSUB;
 if $tmp is zero:
   return $r;
 end
$tmp = $k - TK_FMUL;
 if $tmp is zero:
   return $r;
 end
$tmp = $k - TK_FDIV;
 if $tmp is zero:
   return $r;
 end
$tmp = $k - TK_FREM;
 if $tmp is zero:
   return $r;
 end
$tmp = $k - TK_PIPE;
 if $tmp is zero:
   return $r;
 end
$tmp = $k - TK_CARET;
 if $tmp is zero:
   return $r;
 end
$tmp = $k - TK_SHL;
 if $tmp is zero:
   return $r;
 end
$tmp = $k - TK_SHR;
 if $tmp is zero:
   return $r;
```

```
    end
  $tmp = $k - TK_SHLU;
    if $tmp  is  zero:
      return $r;
    end
  $tmp = $k - TK_SHRU;
    if $tmp  is  zero:
      return $r;
    end
  $tmp = $k - TK_2ADDU;
    if $tmp  is  zero:
      return $r;
    end
  $tmp = $k - TK_4ADDU;
    if $tmp  is  zero:
      return $r;
    end
  $tmp = $k - TK_8ADDU;
    if $tmp  is  zero:
      return $r;
    end
  $tmp = $k - TK_16ADDU;
    if $tmp  is  zero:
      return $r;
    end
  $tmp = $k - TK_AMP;
    if $tmp  is  zero:
      return $r;
    end
  $r = 0;
    return $r;
end
```

⟨ ***Parser procedures*** ⟩ +≡

```
def IsKwBinop($v) -> $r:
  register $tmp;
  # Named binary-op keywords occupy a contiguous range
  # KW_ANDN(60)..KW_UNEDOT(77) in the keyword table.
  $r = 0;
  $tmp = $v - 60;
    if $tmp  is  neg:
      return $r;
    end
  $tmp = $v - 78;
    if $tmp  is  nonneg:
      return $r;
    end
  $r = 1;
    return $r;
end
```

`ParseRHSExpr` parses one RHS expression: an atom optionally followed by one binary operator and a second atom. The resulting node is either the bare atom or an `NK_BINOP` with `ND_A`=lhs, `ND_B`=rhs, `ND_C`=operator code, `ND_D`=0 for a token-encoded operator or 1 for a keyword-encoded one.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseRHSExpr() -> $r:
  register $lhs, $rhs, $k, $tmp, $node, $isKw, $op, $val;
 $lhs = call ParseAtom();
 $isKw = 0;


 $op = 0;
 $k = call Peek();
 $tmp = call IsTokBinop($k);
 goto PRHTok if $tmp is nonzero;
 $tmp = $k - TK_KW;
 if $tmp is nonzero:
   return $lhs;
 end
 $val = call TokValue($cur_tok);
 $tmp = call IsKwBinop($val);
 if $tmp is zero:
   return $lhs;
 end
 $op = $val;
 $isKw = 1;
 call Advance();
 goto PRHBin;
PRHTok:
 $op = $k;
 call Advance();
PRHBin:
 $rhs = call ParseAtom();
 $node = call NewNode(NK_BINOP);
 call NodeSet($node, ND_A, $lhs);
 call NodeSet($node, ND_B, $rhs);
 call NodeSet($node, ND_C, $op);
 call NodeSet($node, ND_D, $isKw);
 return $node;
PRHDone:
 return $lhs;
end
```

`ParseAssign` handles the register-led statement forms `$dst = RHS;` (single assignment) and `$dst[, $dst2, ...] = call Proc(args);` (multi-return call). A comma-separated register-reference list is collected first, the `=` consumed, and then the head token decides: `call` keyword means a call-assign, anything else means a single-register assignment (in which case the register list must have exactly one element).

Layout of `NK_ASSIGN`: `ND_A` = single dst (reg-ref node), `ND_B` = RHS expression. Layout of `NK_CALLASN`: `ND_A` = dst-list head (linked through `ND_NEXT`), `ND_B` = callee label (for direct `call Proc(...)`) or indirect reg-ref (for `call [$fp](...)`), `ND_C` = arg-list head, `ND_D` = 1 if indirect else 0. Layout of `NK_WYDESET` (produced for `$dst ?= imm` and `$dst orh= imm`-style statements): `ND_A` = single dst, `ND_B` = RHS expression, `ND_C` = wyde-op code (0 for `?=`, else the `TK_WYDEOP` value), `ND_D` = 1 for a wyde-op form, 0 for `?=`. Only single-register LHS is allowed.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseAssign() -> $r:
  register $head, $tail, $item, $m, $k, $tmp, $node, $single, $opcode;
 $head = 0;
 $tail = 0;


 $item =  call ParseRegRef();
 $head = $item;
 $tail = $item;
 $single = $item;
PAsnLoop:
 $m =  call MatchKind(TK_COMMA);
  goto PAsnEnd  if $m  is  zero;
  call Advance();
 $item =  call ParseRegRef();
  call NodeSet($tail, ND_NEXT, $item);
 $tail = $item;
 $single = 0;
  goto PAsnLoop;
PAsnEnd:
 $k =  call Peek();
 $tmp = $k - TK_EQUALS;
  goto PAsnEq    if $tmp  is  zero;
 $tmp = $k - TK_ZASSIGN;
  goto PAsnZasn  if $tmp  is  zero;
 $tmp = $k - TK_WYDEOP;
  goto PAsnWyde  if $tmp  is  zero;
  # Default: fall through to Expect(=) to produce the standard error.
  call Expect(TK_EQUALS);
PAsnEq:
  call Advance();
 $m =  call MatchKw(KW_CALL);
  goto PAsnCall  if $m  is  nonzero;
 $m =  call MatchKw(KW_CSWAP);
  goto PAsnCswap  if $m  is  nonzero;
  # Single-register assignment only---reject multi-LHS without call.
  goto PAsnOne  if $single  is  nonzero;
  call ErrCurTokHalt(&ErrParseAsnLHS);
PAsnOne:
 $r =  call NewNode(NK_ASSIGN);
  call NodeSet($r, ND_A, $single);
 $node =  call ParseRHSExpr();
  call NodeSet($r, ND_B, $node);
  return  call ParseStmtPostfixIf($r);
PAsnCall:
 $r =  call ParseCallTail(NK_CALLASN);
  call NodeSet($r, ND_A, $head);
  call Expect(TK_SEMI);
  return $r;
PAsnCswap:
  goto PAsnCswapOk  if $single  is  nonzero;
  call ErrCurTokHalt(&ErrParseAsnLHS);
PAsnCswapOk:
  return  call ParseCswap($single);
PAsnZasn:
  goto PAsnZasnOk  if $single  is  nonzero;
  call ErrCurTokHalt(&ErrParseAsnLHS);
PAsnZasnOk:
  call Advance();
 $r =  call NewNode(NK_WYDESET);
  call NodeSet($r, ND_A, $single);
```

```
    $node = call ParseRHSExpr();
     call NodeSet($r, ND_B, $node);
     call NodeSet($r, ND_C, 0);
     call NodeSet($r, ND_D, 0);
     return call ParseStmtPostfixIf($r);
    PAsnWyde:
     goto PAsnWydeOk if $single is nonzero;
     call ErrCurTokHalt(&ErrParseAsnLHS);
    PAsnWydeOk:
    $opcode = call TokValue($cur_tok);
     call Advance();
    $r = call NewNode(NK_WYDESET);
     call NodeSet($r, ND_A, $single);
    $node = call ParseRHSExpr();
     call NodeSet($r, ND_B, $node);
     call NodeSet($r, ND_C, $opcode);
     call NodeSet($r, ND_D, 1);
     return call ParseStmtPostfixIf($r);
  end
```

`ParseCallVoid` parses `call Proc(args);` or `call [$fp](args);` with no return-value list; it is reached from the main statement dispatcher on a bare `call` keyword. `ParseCallTail` is factored out so it can be shared with the call-assign path in `ParseAssign` and the `return call` path in `ParseReturnExpr`; it consumes the `call` keyword and everything up to but not including the trailing `;`, which each caller handles itself. The caller provides the target node kind (`NK_CALL` for void, `NK_CALLASN` for call-assign) and is responsible for filling `ND_A` with the destination list when appropriate.

⟨ ***Parser procedures*** ⟩ +≡

```
  def ParseCallVoid() -> $r:
   $r = call ParseCallTail(NK_CALL);
    call Expect(TK_SEMI);
    return $r;
  end
```

⟨ ***Parser procedures*** ⟩ +≡

```
  def ParseCallTail($nk) -> $r:
    register $m, $tmp, $val, $len, $fp, $args;
    call ExpectKw(KW_CALL);
   $r = call NewNode($nk);
   $m = call MatchKind(TK_LBRACKET);
    goto PCtInd if $m is nonzero;
   $val = call TokValue($cur_tok);
   $len = call TokLen($cur_tok);
    call Expect(TK_LABEL);
    # For direct call, ND_B holds an NK_LABELREF node wrapping the name.
   $tmp = call NewNode(NK_LABELREF);
    call NodeSet($tmp, ND_A, $val);
    call NodeSet($tmp, ND_B, $len);
    call NodeSet($r, ND_B, $tmp);
    call NodeSet($r, ND_D, 0);
    goto PCtArgs;
   PCtInd:
    call Expect(TK_LBRACKET);
   $fp = call ParseRegRef();
```

```
    call Expect(TK_RBRACKET);
    call NodeSet($r, ND_B, $fp);
    call NodeSet($r, ND_D, 1);
  PCtArgs:
  $args =  call ParseArgList();
    call NodeSet($r, ND_C, $args);
    return $r;
end
```

`ParseArgList` parses a parenthesised, comma-separated list of call arguments and returns the head of a linked chain of argument nodes (or 0 for an empty list). Each argument is one of: a register reference, an immediate literal, a label (treated uniformly as an `NK_LABELREF`—the generator decides whether it is a const alias or a symbol), or an `&Label` (`NK_GETA`).

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseArgList() -> $r:
  register $head, $tail, $item, $m;
  call Expect(TK_LPAREN);
 $head = 0;
 $tail = 0;
 $m =  call MatchKind(TK_RPAREN);
  goto PALNonEmpty  if $m  is  zero;
  call Advance();
  goto PALDone;
 PALNonEmpty:
 $item =  call ParseArgItem();
 $head = $item;
 $tail = $item;
 PALLoop:
 $m =  call MatchKind(TK_COMMA);
  goto PALEnd  if $m  is  zero;
  call Advance();
 $item =  call ParseArgItem();
  call NodeSet($tail, ND_NEXT, $item);
 $tail = $item;
  goto PALLoop;
 PALEnd:
  call Expect(TK_RPAREN);
 PALDone:
  return $head;
end
```

When the next token is a string literal, `ParseArgItem` treats it as an inline string-literal argument: it allocates a fresh hidden data-segment label, builds an `NK_GETA` node referring to that label, and queues a synthesised `byte LABEL "...", 0;` declaration for splicing into the data segment by `ParseProgram`. This lets a caller write `call ErrHalt("oops\n");` instead of having to predeclare every error message as a separate top-level `byte` directive.

⟨ ***Parser procedures*** ⟩ +≡

```
def PoolAppendInt($n):
  register $count, $digit, $tmp, $ptr;
  goto PAIWork  if $n  is  nonzero;
  call PoolAppendByte('0');


  return;
 PAIWork:
 $count = 0;
 PAILoop:
  goto PAIEmit  if $n  is  zero;
 $tmp = $n / 10;
 $digit = $tmp * 10;
 $digit = $n - $digit;
 $digit = $digit + '0';
 $ptr = $IntBuf + $count;
 [$ptr].bu = $digit;
 $count = $count + 1;
 $n = $tmp;
  goto PAILoop;
 PAIEmit:
 PAIEmitLoop:
  goto PAIEDone  if $count  is  zero;
 $count = $count - 1;
 $ptr = $IntBuf + $count;
 $digit = [$ptr].bu;
  call PoolAppendByte($digit);
  goto PAIEmitLoop;
 PAIEDone:
  return;
end
```

⟨ ***Parser procedures*** ⟩ +≡

```
def AllocInlineStrLabel() -> $r:
  register $ctr;
 $r = $str_end;
  call PoolAppendByte('_');
  call PoolAppendByte('_');
  call PoolAppendByte('I');
  call PoolAppendByte('n');
  call PoolAppendByte('l');
  call PoolAppendByte('i');
  call PoolAppendByte('n');
  call PoolAppendByte('e');
  call PoolAppendByte('S');
  call PoolAppendByte('t');
  call PoolAppendByte('r');
 $ctr =  call InlineStrCtrNext();
  call PoolAppendInt($ctr);
  return $r;
end
```

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseArgItem() -> $r:
  register $k, $tmp, $val, $len, $loff, $llen;
  register $strnode, $zeronode, $list, $datad, $chain;
 $k =  call Peek();


 $tmp = $k - TK_STRING;
  goto PAIStr  if $tmp  is  zero;
  return  call ParseAtom();
PAIStr:
 $val =  call TokValue($cur_tok);
 $len =  call TokLen($cur_tok);
  call Advance();
 $loff =  call AllocInlineStrLabel();
 $llen = $str_end - $loff;
 $r =  call NewNode(NK_GETA);
  call NodeSet($r, ND_A, $loff);
  call NodeSet($r, ND_B, $llen);
 $strnode =  call NewNode(NK_STRING);
  call NodeSet($strnode, ND_A, $val);
  call NodeSet($strnode, ND_B, $len);
 $zeronode =  call NewNode(NK_IMM);
  call NodeSet($strnode, ND_NEXT, $zeronode);
 $list = $strnode;
 $datad =  call NewNode(NK_DATADIR);
  call NodeSet($datad, ND_C, KW_BYTE);
  call NodeSet($datad, ND_A, $loff);
  call NodeSet($datad, ND_B, $llen);
  call NodeSet($datad, ND_D, $list);
 $chain =  call InlineStrChainGet();
 $chain =  call NodeAppend($chain, $datad);
  call InlineStrChainSet($chain);
  return $r;
end
```

Load and store address grammar. Inside the brackets of `[addr].sz`, `addr` is `base` or `base + offset` or `base - offset`, where both `base` and `offset` are parsed as general atoms. A subtracted offset is wrapped in an `NK_UNOP(TK_MINUS)` so the generator sees a uniform "additive offset" shape.

`ParseAddrPair` fills `ND_A`/`ND_B` of its argument node with the base and (optional) offset. Both `ParseLoad` and `ParseStore` delegate the address walk to it, so the more elaborate forms (label-based `Label[off].sz` and register-indexed `$base[$idx].sz`) can be added later in the same helper without touching the callers.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseAddrPair($node):
  call ParseAddrInto($node, ND_A, ND_B);
  return;
end
```

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseAddrInto($node, $basefield, $offfield):
  register $base, $off, $neg, $k, $tmp;
 $base =  call ParseAtom();
  call NodeSet($node, $basefield, $base);
 $k =  call Peek();
 $tmp = $k - TK_PLUS;
  goto PAPPlus   if $tmp  is  zero;
 $tmp = $k - TK_MINUS;
  goto PAPMinus  if $tmp  is  zero;
  return;
 PAPPlus:
  call Advance();
 $off =  call ParseAtom();
  call NodeSet($node, $offfield, $off);
  return;
 PAPMinus:
  call Advance();
 $off =  call ParseAtom();
 $neg =  call NewNode(NK_UNOP);
  call NodeSet($neg, ND_A, $off);
  call NodeSet($neg, ND_C, TK_MINUS);
  call NodeSet($node, $offfield, $neg);
  return;
end
```

`MaybeIndexedLoad` implements the base-indexed memref shorthands `$base[$idx].sz` and `Label[off].sz` in rvalue contexts. Given an already-parsed base node (`NK_REG`, `NK_REGNAME`, or `NK_LABELREF`), it peeks at the next token: if it is `[`, the base is wrapped in an `NK_LOAD` node with the bracketed atom as its `ND_B` offset and the trailing `.sz` size code in `ND_C`. Otherwise the base is returned unchanged. Callers in `ParseAtom` pass every atom kind that can legally stand as a load base.

⟨ ***Parser procedures*** ⟩ +≡

```
def MaybeIndexedLoad($base) -> $r:
  register $k, $tmp, $idx, $size, $load;
 $r = $base;
 $k =  call Peek();
 $tmp = $k - TK_LBRACKET;
  goto MILDone  if $tmp  is  nonzero;
  call Advance();
 $idx =  call ParseAtom();
  call Expect(TK_RBRACKET);
 $size =  call TokValue($cur_tok);
  call Expect(TK_SIZE);
 $load =  call NewNode(NK_LOAD);
  call NodeSet($load, ND_A, $base);
  call NodeSet($load, ND_B, $idx);
  call NodeSet($load, ND_C, $size);
 $r = $load;
 MILDone:
  return $r;
end
```

`ParseStoreShort` parses a shorthand store statement, either `$base[$idx].sz = rhs;` or `Label[off].sz = rhs;`. The base is parsed directly here rather than via `ParseAtom`, since `ParseAtom` would

already have eaten the `[...]` and returned a load node. The resulting `NK_STORE` has `ND_A` = base, `ND_B` = offset/index, `ND_C` = size code, `ND_D` = RHS expression, matching the layout produced by `ParseStore`.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseStoreShort() -> $r:
  register $base, $idx, $size, $rhs, $val, $len, $k, $tmp;
 $k = call Peek();
 $tmp = $k - TK_LABEL;


 goto PStSLbl if $tmp is zero;
 $base = call ParseRegRef();
 goto PStSBr;
PStSLbl:
 $val = call TokValue($cur_tok);
 $len = call TokLen($cur_tok);
 call Advance();
 $base = call NewNode(NK_LABELREF);
 call NodeSet($base, ND_A, $val);
 call NodeSet($base, ND_B, $len);
PStSBr:
 call Expect(TK_LBRACKET);
 $idx = call ParseAtom();
 call Expect(TK_RBRACKET);
 $size = call TokValue($cur_tok);
 call Expect(TK_SIZE);
 call Expect(TK_EQUALS);
 $rhs = call ParseRHSExpr();
 call Expect(TK_SEMI);
 $r = call NewNode(NK_STORE);
 call NodeSet($r, ND_A, $base);
 call NodeSet($r, ND_B, $idx);
 call NodeSet($r, ND_C, $size);
 call NodeSet($r, ND_D, $rhs);
 return $r;
end
```

`ParseLoad` parses an rvalue load expression `[addr].sz`. The size suffix is a single `TK_SIZE` token whose `TK_VALUE` carries a size-code; the code is stored directly in `ND_C` and interpreted by the generator.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseLoad() -> $r:
  register $size;
  call Expect(TK_LBRACKET);
 $r = call NewNode(NK_LOAD);
  call ParseAddrPair($r);
  call Expect(TK_RBRACKET);
 $size = call TokValue($cur_tok);
  call Expect(TK_SIZE);
  call NodeSet($r, ND_C, $size);
  return $r;
end
```

`ParseStore` parses the statement form `[addr].sz = rhs;`. The RHS is a general runtime

expression (`ParseRHSExpr`), which is the same expression grammar used on the right-hand side of a register assignment. The address base goes in `ND_A`, the offset (or 0) in `ND_B`, the size code in `ND_C`, and the RHS node in `ND_D`.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseStore() -> $r:
  register $size, $rhs;
  call Expect(TK_LBRACKET);
 $r =  call NewNode(NK_STORE);
  call ParseAddrPair($r);
  call Expect(TK_RBRACKET);
 $size =  call TokValue($cur_tok);
  call Expect(TK_SIZE);
  call NodeSet($r, ND_C, $size);
  call Expect(TK_EQUALS);
 $rhs =  call ParseRHSExpr();
  call NodeSet($r, ND_D, $rhs);
  call Expect(TK_SEMI);
  return $r;
end
```

`ParseUnaryOp` parses a keyword unary-operator application of the form `op operand` or `op[rmode] operand`. The current token is a `TK_UNARYOP` whose `TK_VALUE` is the op code (1..8). An optional rounding-mode bracket—`[nearest]`, `[zero]`, `[up]`, `[down]`—may follow for floating-point forms like `sflot`. The operand is a general atom (no binary operator; the user parenthesises by introducing a temporary if needed).

Layout: `NewNodeExtra(NK_UNOP, 1)`. `ND_A` = operand node, `ND_C` = op code, `ND_D` = 1 (signalling keyword-encoded op to the generator), and the single extra octa at `NODE_HEADER_SIZE + 0` holds the rounding-mode keyword code, or 0 when absent.

⟨ ***Parser procedures*** ⟩ +≡

```
def ParseUnaryOp() -> $r:
  register $op, $k, $tmp, $m, $rmode, $operand;
 $op =  call TokValue($cur_tok);
  call Advance();


 $rmode = 0;
 $k =  call Peek();
 $tmp = $k - TK_LBRACKET;
  goto PUoNoRm  if $tmp  is  nonzero;
  call Advance();
 $m =  call MatchKw(KW_NEAREST);
  goto PUoRmOk  if $m  is  nonzero;
 $m =  call MatchKw(KW_ZERO);
  goto PUoRmOk  if $m  is  nonzero;
 $m =  call MatchKw(KW_UP);
  goto PUoRmOk  if $m  is  nonzero;
 $m =  call MatchKw(KW_DOWN);
  goto PUoRmOk  if $m  is  nonzero;
  call ErrCurTokHalt("plmmix-compiler: parse error: expected rounding mode in brackets\n");
PUoRmOk:
 $rmode =  call TokValue($cur_tok);
  call Advance();
  call Expect(TK_RBRACKET);
PUoNoRm:
```

```
  $operand = call ParseAtom();
  $r = call NewNodeExtra(NK_UNOP, 1);
  call NodeSet($r, ND_A, $operand);
  call NodeSet($r, ND_C, $op);
  call NodeSet($r, ND_D, 1);
  call NodeSet($r, NODE_HEADER_SIZE, $rmode);
  return $r;
end
```

Parser error messages introduced in this section.

⟨ ***Data segment*** ⟩ +≡

```
byte ErrParseAsnLHS  "plmmix-compiler: parse error: multi-register LHS allowed only with
 `call'\n", 0;
```

Parser error messages introduced in this section.

⟨ ***Data segment*** ⟩ +≡

```
byte ErrParseRegRef   "plmmix-compiler: parse error: expected register (`$N' or `$name')\n",
 0;
byte ErrAliasFull     "plmmix-compiler: generator alias table full\n", 0;
byte StrSymHALT       "HALT", 0;
byte StrSymFOPEN      "FOPEN", 0;
byte StrSymFCLOSE     "FCLOSE", 0;
byte StrSymFREAD      "FREAD", 0;
byte StrSymFGETS      "FGETS", 0;
byte StrSymFGETWS     "FGETWS", 0;
byte StrSymFWRITE     "FWRITE", 0;
byte StrSymFPUTS      "FPUTS", 0;
byte StrSymFPUTWS     "FPUTWS", 0;
byte StrSymFSEEK      "FSEEK", 0;
byte StrSymFTELL      "FTELL", 0;
byte StrSymSTDIN      "STDIN", 0;
byte StrSymSTDOUT     "STDOUT", 0;
byte StrSymSTDERR     "STDERR", 0;
```

22. Generator: design overview.

The generator turns the AST into an MMIXAL source file in a single forward pass. Three properties of the target language drive its shape, and each one motivates state that might otherwise look incidental.

Flat label namespace. MMIXAL has a single flat label namespace: a label defined inside one procedure collides at assembly time with the same label defined inside another, even though in PL/MMIX they are independent proc-local labels. The generator therefore mangles a proc-local label only when its name actually collides — prefixing every local label with its enclosing proc would be unreadable and would defeat the feature. `ScanDupLabels` performs the cross-procedure pre-pass that counts how many times each label name appears across all `def` bodies, recording the result through `DupLabelBump` as a `SYM_DUP_COUNT` field in the unified symbol table. During emit, a label whose count is 1 is written verbatim; a label whose count is greater than 1 is prefixed with the enclosing proc name to disambiguate. The per-proc scratch `$ProcLabelBuf` is reset at

the start of each `def` body, used to collect that body's labels, and consulted by `ProcLabelMode` during emission to decide whether the current label is local to this proc; the duplicate flag is read directly from the global symbol table via `IsDupLabel`.

Forward references and emit shape. PL/MMIX permits an instruction to reference a data label declared later in the source. Worse, the MMIXAL emit shape for a name depends on what kind of declaration it ultimately resolves to:

- • a plain `byte`/`reserve` label is loaded through the data-segment base register;
- • a name bound by `const x = expr` is replaced inline by an MMIXAL `IS` equivalence and may be emitted as an immediate;
- • a name allocated with `greg` becomes a direct register reference, not a load;
- • a label whose address is taken anywhere in the program needs a `GETA`-friendly form.

We cannot tell which case applies without first knowing every top-level declaration. Hence three pre-passes run before the AST walk emits any code: `ScanTopDataLabels` interns every top-level `byte`/`reserve`/`const` name into the symbol table with the `SYM_F_DATA` flag (and marks likely-address-taken names from textual cues), `ScanTopConstDefs` attaches the const-expression handle and the `SYM_F_CONST` flag to the matching entry, and `ScanAddrTakenTop` walks the program looking for actual address-taking uses and ORs `SYM_F_ADDR_TAKEN` onto those entries. `greg`-allocated names get the `SYM_F_GREG` flag, set incidentally as `greg` declarations are emitted. Throughout emission, the predicates `IsDataLabel`, `LookupConstExpr`, `IsGregName`, and `IsDataLabelAddrTaken` are thin wrappers over `SymLookup` that test the appropriate flag bit.

Register allocator. PL/MMIX `register $name` declarations and procedure parameters consume specific `$N` registers, and the generator also synthesises scratch registers for compound expressions. To pick a free `$N` without colliding with a user binding the generator maintains two 256-byte bitmaps, `$GlobalRegUsed` and `$LocalRegUsed`, one per scope. `MarkRegUsed` sets a bit when an alias is bound; `AllocFreeReg` linearly scans the bitmap to find the next available register; `AliasReset` and `ClearRegUsed` restart the local map at the top of every `def`. Register `$17` is pre-marked at reset time because MMIX reserves it as a scratch.

Tables, in summary. `$SymBuf` is the unified generator symbol table: one entry per unique normalized name, with flag bits recording which kinds of declaration apply (`SYM_F_DATA`, `SYM_F_CONST`, `SYM_F_GREG`, `SYM_F_ADDR_TAKEN`), the const-binding's AST handle, and the cross-procedure label-occurrence count. Lookups go through an open-addressed hash table `$SymHashTbl` (size `SYM_HASH_SIZE`, linear probing, djb2 hash) so that `IsDataLabel`, `IsGregName` and friends are expected $O(1)$ rather than scanning every entry. All access funnels through `SymLookup` and `SymIntern`, which strip a leading `$` from the input so `greg`-style names match their bare-label equivalents. `$ProcLabelBuf` is the per-procedure label scratch used by `ProcLabelMode`. `$GlobalRegUsed` and `$LocalRegUsed` are the register-allocator bitmaps. `$GlobalAliasBuf` and `$LocalAliasBuf` map `register $name` aliases to `$N`. All of these are reset at the top of `Generate` via `AliasReset`, then populated by the pre-pass calls before the AST walk begins.

23. Generator: top-level skeleton.

`Generate` is the entry point called from `main` after `ParseProgram` has filled the AST and left its root handle in `$ast_root`. It resets generator-global state, emits the driver banner, performs the top-level pre-scans that collect duplicate labels, constant definitions, and address-taken data labels, then walks the `NK_PROGRAM.ND_A` list (linked through `ND_NEXT`) and dispatches each child to `GenTopItem`. The later generator sections refine that walk: the driver-banner section emits

the banner, the procedure-body section handles instruction selection, and the later control-flow helper code supplies the synthetic-label and branch helpers shared by loops and conditionals.

⟨ ***Generator procedures*** ⟩ ≡

```
def Generate():
  register $root, $item, $tmp;
  call AliasReset();
  call GenBanner();


 $root = $ast_root;
  if $root  is  zero:
    return;
  end
 $item =  call NodeGet($root, ND_A);
  call ScanDupLabels($item);
  call ScanTopDataLabels($item);
  call ScanTopConstDefs($item);
  call ScanAddrTakenTop($item);
 GLoop:
  goto GFinal  if $item  is  zero;
 $tmp = &StrNodeLine;
  call AppendCStr($tmp);
 $tmp =  call NodeGet($item, ND_LINE);
  call AppendInt($tmp);
  call AppendByte('\n');
  call GenTopItem($item);
 $item =  call NodeGet($item, ND_NEXT);
  goto GLoop;
 GFinal:
  # Finalization state machine for |$main_emitted|:
  #   0  = no code-segment location or Main emitted yet
  #  -1  = code location emitted (for early procedures), Main pending
  #   1  = Main emitted
  # When the user wrote an explicit |loc| directive (raw/kernel mode),
  # do not synthesize |LOC #100| or |Main IS @|: the user owns placement.
 $tmp = $main_emitted - 1;
  if  likely $tmp  is  zero:
    return;
  end
  if $user_loc_seen  is  nonzero:
    return;
  end
  if $main_emitted  is  neg:
   $tmp = &StrMainIs;
    call AppendCStr($tmp);
   $main_emitted = 1;
    return;
  end
 $tmp = &StrSegCode;
  call AppendCStr($tmp);
 $tmp = &StrMainIs;
  call AppendCStr($tmp);
 $main_emitted = 1;
  return;
end
```

Alias helpers for the generator.

⟨ ***Generator procedures*** ⟩ ≡

```
def ClearRegUsed($base):
  register $idx, $tmp;
 $idx = 0;
 CRULoop:
 $tmp = $idx - 256;
  goto CRUDone  if $tmp  is  nonneg;
 [$base + $idx].bu = 0;
 $idx = $idx + 1;
  goto CRULoop;
 CRUDone:
  return;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def AliasReset():
 $GlobalAliasCnt = 0;
 $LocalAliasCnt = 0;
 $SymCnt = 0;
 $ProcLabelCnt = 0;
 $gen_in_proc = 0;
  call SymHashClear();
  call ClearRegUsed($GlobalRegUsed);
  call ClearRegUsed($LocalRegUsed);
  call MarkRegUsed($GlobalRegUsed, 17);
  call MarkRegUsed($LocalRegUsed, 17);
  return;
end
```

Generator symbol-table primitives. The pre-passes and emit-time queries described in the design overview all share the single `$SymBuf` table; these routines (and their thin wrappers below) are the only places that reach into the entry layout. Lookups go through an open-addressed hash table `$SymHashTbl` indexed by a djb2 hash of the (normalized) name, with linear probing on collision; this turns what was a linear scan over thousands of entries into expected $O(1)$ work. The leading `$` that `greg`-allocated names carry in their string-pool entry is stripped at the boundary so that `SymLookup("$X", 2)` matches an entry stored under the bare name `X`.

⟨ ***Generator procedures*** ⟩ ≡

```
def SymHash($off, $len) -> $r:
  register $h, $end, $ch, $tmp, $shifted;
 $h = 5381;
 $end = $off + $len;
 SHashLoop:
 $tmp = $off - $end;
  goto SHashDone  if $tmp  is  nonneg;
 $ch = [$StringPool + $off].bu;
  # |$h * 33| as |($h << 5) + $h|.  MUL by a small constant on MMIX
  # costs ~10 oops; this is the inner loop of every symbol lookup.
 $shifted = $h << 5;
 $h = $shifted + $h;
 $h = $h + $ch;
 $off = $off + 1;
  goto SHashLoop;
 SHashDone:
```

```
  $r = $h;
   return $r;
end
```

⟨ **Generator procedures** ⟩ ≡

```
def SymHashClear():
   register $idx, $end, $tmp;
  $idx = 0;
  $end = SYM_HASH_SIZE * 8;
 SHCLoop:
  $tmp = $idx - $end;
   goto SHCDone  if $tmp  is  nonneg;
 [$SymHashTbl + $idx].o = 0;
  $idx = $idx + 8;
   goto SHCLoop;
 SHCDone:
   return;
end
```

⟨ **Generator procedures** ⟩ ≡

```
def SymLookup($off, $len) -> $r:
   register $hash, $slot, $slotaddr, $ent, $eoff, $elen, $tmp, $p1, $p2, $ch;
  $r = 0;
   if $len  is  nonzero:


     $p2 = $StringPool + $off;
     $ch = [$p2].bu;
     $tmp = $ch - '$';
      if $tmp  is  zero:
       $off = $off + 1;
       $len = $len - 1;
      end
   end
  $hash =  call SymHash($off, $len);
  $slot = $hash & SYM_HASH_MASK;
 SLProbe:
  $slotaddr = $slot << 3;             # * 8
  $slotaddr = $slotaddr + $SymHashTbl;
  $ent = [$slotaddr].o;
   if $ent  is  zero:
     return $r;
   end
  $elen = [$ent + SYM_NAME_LEN].o;
  $tmp = $elen - $len;
   goto SLNext  if $tmp  is  nonzero;
  $eoff = [$ent + SYM_NAME_OFF].o;
  $p1 = $StringPool + $eoff;
  $p2 = $StringPool + $off;
  $tmp =  call StrEq($p1, $elen, $p2, $len);
   if $tmp  is  nonzero:
    $r = $ent;
     return $r;
   end
 SLNext:
  $slot = $slot + 1;
```

```
  $slot = $slot & SYM_HASH_MASK;
   goto SLProbe;
 end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def SymIntern($off, $len) -> $r:
  register $hash, $slot, $slotaddr, $ent, $eoff, $elen, $tmp, $p1, $p2, $ch;
  if $len  is  nonzero:
   $p2 = $StringPool + $off;


   $ch = [$p2].bu;
   $tmp = $ch - '$';
    if $tmp  is  zero:
     $off = $off + 1;
     $len = $len - 1;
    end
  end
 $hash =  call SymHash($off, $len);
 $slot = $hash & SYM_HASH_MASK;
 SIProbe:
 $slotaddr = $slot << 3;              # * 8
 $slotaddr = $slotaddr + $SymHashTbl;
 $ent = [$slotaddr].o;
  if $ent  is  zero:
    goto SIEmpty;
  end
 $elen = [$ent + SYM_NAME_LEN].o;
 $tmp = $elen - $len;
  goto SINext  if $tmp  is  nonzero;
 $eoff = [$ent + SYM_NAME_OFF].o;
 $p1 = $StringPool + $eoff;
 $p2 = $StringPool + $off;
 $tmp =  call StrEq($p1, $elen, $p2, $len);
  if $tmp  is  nonzero:
   $r = $ent;
    return $r;
  end
 SINext:
 $slot = $slot + 1;
 $slot = $slot & SYM_HASH_MASK;
  goto SIProbe;
 SIEmpty:
 $tmp = $SymCnt - SYM_MAX;
  if $tmp  is  nonneg:
    call ErrHalt(&ErrAliasFull);
  end
 $ent = $SymCnt * SYM_ENTRY_SIZE;
 $ent = $ent + $SymBuf;
 [$ent + SYM_NAME_OFF].o = $off;
 [$ent + SYM_NAME_LEN].o = $len;
 [$ent + SYM_FLAGS].o = 0;
 [$ent + SYM_CONST_EXPR].o = 0;
 [$ent + SYM_DUP_COUNT].o = 0;
 $SymCnt = $SymCnt + 1;
 [$slotaddr].o = $ent;
 $r = $ent;
  return $r;
end
```

⟨ **Generator procedures** ⟩ ≡

```
def SymSetFlag($ent, $mask):
  register $flags;
 $flags = [$ent + SYM_FLAGS].o;
 $flags = $flags | $mask;
 [$ent + SYM_FLAGS].o = $flags;
  return;
end
```

⟨ **Generator procedures** ⟩ ≡

```
def SymHasFlag($ent, $mask) -> $r:
  register $flags, $tmp;
 $r = 0;
  if $ent  is  zero:
    return $r;
  end
 $flags = [$ent + SYM_FLAGS].o;
 $tmp = $flags & $mask;
  if $tmp  is  nonzero:
   $r = 1;
  end
  return $r;
end
```

⟨ **Generator procedures** ⟩ ≡

```
def RegisterConstDef($off, $len, $expr):
  register $ent;
 $ent =  call SymIntern($off, $len);
  call SymSetFlag($ent, SYM_F_DATA);
  call SymSetFlag($ent, SYM_F_CONST);
 [$ent + SYM_CONST_EXPR].o = $expr;
  return;
end
```

⟨ **Generator procedures** ⟩ ≡

```
def LookupConstExpr($off, $len) -> $r:
  register $ent, $tmp;
 $r = 0;
 $ent =  call SymLookup($off, $len);
  if $ent  is  zero:
    return $r;
  end
 $tmp =  call SymHasFlag($ent, SYM_F_CONST);
  if $tmp  is  zero:
    return $r;
  end
 $r = [$ent + SYM_CONST_EXPR].o;
  return $r;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def RegisterGregName($off, $len):
  register $ent;
 $ent = call SymIntern($off, $len);
  call SymSetFlag($ent, SYM_F_GREG);
  return;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def IsGregName($off, $len) -> $r:
  register $ent;
 $ent = call SymLookup($off, $len);
 $r = call SymHasFlag($ent, SYM_F_GREG);
  return $r;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def IsConstName($off, $len) -> $r:
  register $tmp;
 $r = 0;
 $tmp = call LookupBuiltinImm($off, $len);
  if $tmp is nonzero:
    return 1;
  end
 $tmp = call LookupConstExpr($off, $len);
  if $tmp is nonzero:
   $r = 1;
  end
  return $r;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def IsConstExprNode($node) -> $r:
  register $kind, $tmp;
 $r = 0;
 $kind = call NodeGet($node, ND_KIND);
 $tmp = $kind - NK_IMM;
  goto ICENHit if $tmp is zero;
 $tmp = $kind - NK_LABELREF;
  goto ICENHit if $tmp is zero;
 $tmp = $kind - NK_UNOP;
  goto ICENHit if $tmp is zero;
 $tmp = $kind - NK_BINOP;
  goto ICENHit if $tmp is zero;
  return $r;
 ICENHit:
  return 1;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def EvalConstExpr($node) -> $r:
  register $kind, $tmp, $off, $len, $expr, $lhs, $rhs, $op;
 $kind =  call NodeGet($node, ND_KIND);
 $tmp = $kind - NK_IMM;


  goto ECEImm  if $tmp  is  zero;
 $tmp = $kind - NK_LABELREF;
  goto ECELab  if $tmp  is  zero;
 $tmp = $kind - NK_UNOP;
  goto ECEUn  if $tmp  is  zero;
 $tmp = $kind - NK_BINOP;
  goto ECEBin  if $tmp  is  zero;
  call ErrNodeHalt(&ErrParseExpr, $node);
  return 0;
 ECEImm:
  return  call NodeGet($node, ND_A);
 ECELab:
 $off =  call NodeGet($node, ND_A);
 $len =  call NodeGet($node, ND_B);
 $tmp =  call LookupBuiltinImm($off, $len);
  if $tmp  is  nonzero:
   $r = $tmp - 1;
    return $r;
  end
 $expr =  call LookupConstExpr($off, $len);
  if $expr  is  nonzero:
    return  call EvalConstExpr($expr);
  end
  call ErrNodeHalt(&ErrParseExpr, $node);
  return 0;
 ECEUn:
 $op =  call NodeGet($node, ND_C);
 $tmp = $op - TK_MINUS;
  if $tmp  is  zero:
   $lhs =  call NodeGet($node, ND_A);
   $r =  call EvalConstExpr($lhs);
   $r = - $r;
    return $r;
  end
  call ErrNodeHalt(&ErrParseExpr, $node);
  return 0;
 ECEBin:
 $lhs =  call NodeGet($node, ND_A);
 $rhs =  call NodeGet($node, ND_B);
 $op  =  call NodeGet($node, ND_C);
 $lhs =  call EvalConstExpr($lhs);
 $rhs =  call EvalConstExpr($rhs);
 $tmp = $op - TK_PLUS;
  if $tmp  is  zero:
   $r = $lhs + $rhs;
    return $r;
  end
 $tmp = $op - TK_MINUS;
  if $tmp  is  zero:
   $r = $lhs - $rhs;
    return $r;
  end
```

```
  $tmp = $op - TK_STAR;
   if $tmp  is  zero:
    $r = $lhs * $rhs;
     return $r;
   end
  $tmp = $op - TK_SLASH;
   if $tmp  is  zero:
    $r = $lhs / $rhs;
     return $r;
   end
   call ErrNodeHalt(&ErrParseExpr, $node);
   return 0;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def ScanTopConstDefs($head):
  register $kind, $tmp, $off, $len, $expr;
 STCLoop:
  if $head  is  zero:
    return;
  end
 $kind =  call NodeGet($head, ND_KIND);
 $tmp = $kind - NK_CONST;
  if $tmp  is  zero:
   $off =  call NodeGet($head, ND_A);
   $len =  call NodeGet($head, ND_B);
   $expr =  call NodeGet($head, ND_C);
    call RegisterConstDef($off, $len, $expr);
  end
 $head =  call NodeGet($head, ND_NEXT);
  goto STCLoop;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def ScanTopDataLabels($head):
  register $kind, $tmp, $off, $len, $root;
 $root = $head;
 STDLLoop:


  if $head  is  zero:
    goto STDLMark;
  end
 $kind =  call NodeGet($head, ND_KIND);
 $tmp = $kind - NK_CONST;
  goto STDLName  if $tmp  is  zero;
 $tmp = $kind - NK_DATADIR;
  goto STDLName  if $tmp  is  zero;
 $tmp = $kind - NK_RESERVE;
  goto STDLName  if $tmp  is  zero;
  goto STDLNext;
 STDLName:
 $off =  call NodeGet($head, ND_A);
 $len =  call NodeGet($head, ND_B);
  if $len  is  zero:
    goto STDLNext;
```

```
    end
    call RegisterDataLabel($off, $len);
  STDLNext:
  $head =  call NodeGet($head, ND_NEXT);
    goto STDLLoop;
  STDLMark:
    call ScanAddrTakenTop($root);
    return;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def RegisterDataLabel($off, $len):
    register $ent;
  $ent =  call SymIntern($off, $len);
    call SymSetFlag($ent, SYM_F_DATA);
    return;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def MarkDataLabelAddrTaken($off, $len):
    register $ent;
  $ent =  call SymIntern($off, $len);
    call SymSetFlag($ent, SYM_F_DATA);
    call SymSetFlag($ent, SYM_F_ADDR_TAKEN);
    return;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def IsDataLabel($off, $len) -> $r:
    register $ent;
  $ent =  call SymLookup($off, $len);
  $r =  call SymHasFlag($ent, SYM_F_DATA);
    return $r;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def IsDeclaredDataLabelName($off, $len) -> $r:
    register $ent, $tmp;
  $r = 0;
  $ent =  call SymLookup($off, $len);
  $tmp =  call SymHasFlag($ent, SYM_F_DATA);
    if $tmp  is  zero:
      return $r;
    end
  $tmp =  call SymHasFlag($ent, SYM_F_CONST);
    if $tmp  is  nonzero:
      return $r;
    end
    return 1;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def IsDataLabelAddrTaken($off, $len) -> $r:
  register $ent;
 $ent =  call SymLookup($off, $len);
 $r =  call SymHasFlag($ent, SYM_F_ADDR_TAKEN);
  return $r;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def EmitDataGregName($off, $len):
  register $ptr, $ch;
  call AppendByte('_');
  call AppendByte('_');
  call AppendByte('G');
  call AppendByte('_');
 $ptr = $StringPool + $off;
  if $len  is  zero:
    return;
  end
 $ch = [$ptr].bu;
 $ch = $ch - '$';
  if  likely $ch  is  nonzero:
    call AppendBytes($ptr, $len);
    return;
  end
 $ptr = $ptr + 1;
 $len = $len - 1;
  if  likely $len  is  nonzero:
    call AppendBytes($ptr, $len);
  end
  return;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def EmitDataGregBinding($off, $len):
  register $tmp;
  call EmitDataGregName($off, $len);
 $tmp = &StrGregDir;
  call AppendCStr($tmp);
  call GenPoolName($off, $len);
  call AppendByte('\n');
  call MarkDataLabelAddrTaken($off, $len);
  return;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def ScanAddrTakenExpr($node):
  register $kind, $tmp;
  if $node is zero:
    return;


  end
 $kind = call NodeGet($node, ND_KIND);
 $tmp = $kind - NK_GETA;
 if $tmp is zero:
  $tmp = call NodeGet($node, ND_A);
  $kind = call NodeGet($node, ND_B);
  $node = call IsDeclaredDataLabelName($tmp, $kind);
   if $node is zero:
    $node = call IsDataLabel($tmp, $kind);
     if $node is zero:
       return;
     end
   end
   $node = call LookupConstExpr($tmp, $kind);
   if $node is nonzero:
    $node = call IsConstExprNode($node);
     if $node is nonzero:
       return;
     end
   end
   call MarkDataLabelAddrTaken($tmp, $kind);
   return;
 end
 $tmp = $kind - NK_LABELREF;
 if $tmp is zero:
  $tmp = call NodeGet($node, ND_A);
  $kind = call NodeGet($node, ND_B);
  $node = call IsDeclaredDataLabelName($tmp, $kind);
   if $node is zero:
    $node = call IsDataLabel($tmp, $kind);
     if $node is zero:
       return;
     end
   end
   $node = call LookupConstExpr($tmp, $kind);
   if $node is nonzero:
    $node = call IsConstExprNode($node);
     if $node is nonzero:
       return;
     end
   end
   call MarkDataLabelAddrTaken($tmp, $kind);
   return;
 end
 $tmp = $kind - NK_BINOP;
 if $tmp is zero:
  $tmp = call NodeGet($node, ND_A);
   call ScanAddrTakenExpr($tmp);
  $tmp = call NodeGet($node, ND_B);
   call ScanAddrTakenExpr($tmp);
   return;
 end
 $tmp = $kind - NK_UNOP;
 if $tmp is zero:
  $tmp = call NodeGet($node, ND_A);
```

```
    call ScanAddrTakenExpr($tmp);
    return;
  end
 $tmp = $kind - NK_LOAD;
  if $tmp  is  zero:
   $tmp =  call NodeGet($node, ND_A);
    call ScanAddrTakenExpr($tmp);
   $tmp =  call NodeGet($node, ND_B);
    call ScanAddrTakenExpr($tmp);
    return;
  end
 $tmp = $kind - NK_PAREN;
  if $tmp  is  zero:
   $tmp =  call NodeGet($node, ND_A);
    call ScanAddrTakenExpr($tmp);
    return;
  end
 $tmp = $kind - NK_COND;
  if $tmp  is  zero:
   $tmp =  call NodeGet($node, ND_A);
    call ScanAddrTakenExpr($tmp);
   $tmp =  call NodeGet($node, ND_B);
    call ScanAddrTakenExpr($tmp);
    return;
  end
  return;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def ScanAddrTakenExprList($head):
  register $tmp;
 SATELoop:
  if $head  is  zero:
    return;
  end
  call ScanAddrTakenExpr($head);
 $tmp =  call NodeGet($head, ND_NEXT);
 $head = $tmp;
  goto SATELoop;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def ScanAddrTakenStmt($head):
  register $kind, $tmp, $body, $elif;
  if $head  is  zero:
    return;


  end
 $kind =  call NodeGet($head, ND_KIND);
 $tmp = $kind - NK_BEGIN;
  if $tmp  is  zero:
   $body =  call NodeGet($head, ND_A);
    call ScanAddrTakenStmtList($body);
    return;
  end
```

```
$tmp = $kind - NK_CONST;
 if $tmp  is  zero:
  $tmp =  call NodeGet($head, ND_C);
   call ScanAddrTakenExpr($tmp);
   return;
 end
$tmp = $kind - NK_GREG;
 if $tmp  is  zero:
  $tmp =  call NodeGet($head, ND_C);
   call ScanAddrTakenExpr($tmp);
  $tmp =  call NodeGet($head, ND_D);
   call ScanAddrTakenExpr($tmp);
   return;
 end
$tmp = $kind - NK_ASSIGN;
 if $tmp  is  zero:
  $tmp =  call NodeGet($head, ND_B);
   call ScanAddrTakenExpr($tmp);
   return;
 end
$tmp = $kind - NK_STORE;
 if $tmp  is  zero:
  $tmp =  call NodeGet($head, ND_A);
   call ScanAddrTakenExpr($tmp);
  $tmp =  call NodeGet($head, ND_B);
   call ScanAddrTakenExpr($tmp);
  $tmp =  call NodeGet($head, ND_D);
   call ScanAddrTakenExpr($tmp);
   return;
 end
$tmp = $kind - NK_CALL;
 if $tmp  is  zero:
  $tmp =  call NodeGet($head, ND_C);
   call ScanAddrTakenExprList($tmp);
   return;
 end
$tmp = $kind - NK_CALLASN;
 if $tmp  is  zero:
  $tmp =  call NodeGet($head, ND_C);
   call ScanAddrTakenExprList($tmp);
   return;
 end
$tmp = $kind - NK_GOTO;
 if $tmp  is  zero:
  $tmp =  call NodeGet($head, ND_A);
   call ScanAddrTakenExpr($tmp);
  $tmp =  call NodeGet($head, ND_C);
   call ScanAddrTakenExpr($tmp);
   return;
 end
$tmp = $kind - NK_IF;
 if $tmp  is  zero:
  $tmp =  call NodeGet($head, ND_A);
   call ScanAddrTakenExpr($tmp);
  $body =  call NodeGet($head, ND_B);
   call ScanAddrTakenStmtList($body);
  $elif =  call NodeGet($head, ND_C);
  SATS1ElifLoop:
   if $elif  is  zero:
     goto SATS1ElifDone;
   end
  $tmp =  call NodeGet($elif, ND_A);
```

```
    call ScanAddrTakenExpr($tmp);
   $tmp =  call NodeGet($elif, ND_B);
    call ScanAddrTakenStmtList($tmp);
   $elif =  call NodeGet($elif, ND_NEXT);
    goto SATS1ElifLoop;
   SATS1ElifDone:
   $tmp =  call NodeGet($head, ND_D);
    call ScanAddrTakenStmtList($tmp);
    return;
 end
$tmp = $kind - NK_WHILE;
 if $tmp  is  zero:
   $tmp =  call NodeGet($head, ND_A);
    call ScanAddrTakenExpr($tmp);
   $tmp =  call NodeGet($head, ND_B);
    call ScanAddrTakenStmtList($tmp);
    return;
 end
$tmp = $kind - NK_LOOP;
 if $tmp  is  zero:
   $tmp =  call NodeGet($head, ND_A);
    call ScanAddrTakenStmtList($tmp);
    return;
 end
$tmp = $kind - NK_DO;
 if $tmp  is  zero:
   $tmp =  call NodeGet($head, ND_A);
    call ScanAddrTakenStmtList($tmp);
   $tmp =  call NodeGet($head, ND_B);
    call ScanAddrTakenExpr($tmp);
    return;
 end
$tmp = $kind - NK_RETURN;
 if $tmp  is  zero:
   $tmp =  call NodeGet($head, ND_A);
    call ScanAddrTakenExprList($tmp);
   $tmp =  call NodeGet($head, ND_C);
    call ScanAddrTakenExpr($tmp);
    return;
 end
$tmp = $kind - NK_TRAP;
 if $tmp  is  zero:
   $tmp =  call NodeGet($head, ND_A);
    call ScanAddrTakenExpr($tmp);
   $tmp =  call NodeGet($head, ND_B);
    call ScanAddrTakenExpr($tmp);
   $tmp =  call NodeGet($head, ND_C);
    call ScanAddrTakenExpr($tmp);
    return;
 end
$tmp = $kind - NK_PRELD;
 if $tmp  is  zero:
   $tmp =  call NodeGet($head, ND_B);
    call ScanAddrTakenExpr($tmp);
   $tmp =  call NodeGet($head, ND_C);
    call ScanAddrTakenExpr($tmp);
   $tmp =  call NodeGet($head, ND_D);
    call ScanAddrTakenExpr($tmp);
    return;
 end
$tmp = $kind - NK_SYNC;
 if $tmp  is  zero:
```

```
    $tmp = call NodeGet($head, ND_A);
     call ScanAddrTakenExpr($tmp);
     return;
   end
  $tmp = $kind - NK_TRIP;
   if $tmp is zero:
    $tmp = call NodeGet($head, ND_A);
     call ScanAddrTakenExpr($tmp);
    $tmp = call NodeGet($head, ND_B);
     call ScanAddrTakenExpr($tmp);
    $tmp = call NodeGet($head, ND_C);
     call ScanAddrTakenExpr($tmp);
     return;
   end
  $tmp = $kind - NK_RESUME;
   if $tmp is zero:
    $tmp = call NodeGet($head, ND_A);
     call ScanAddrTakenExpr($tmp);
     return;
   end
  $tmp = $kind - NK_WYDESET;
   if $tmp is zero:
    $tmp = call NodeGet($head, ND_B);
     call ScanAddrTakenExpr($tmp);
     return;
   end
   return;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def ScanAddrTakenStmtList($head):
  register $kind, $tmp, $body, $elif;
 SATSLoop:
  if $head is zero:


    return;
  end
 $kind = call NodeGet($head, ND_KIND);
 $tmp = $kind - NK_BEGIN;
  if $tmp is zero:
   $body = call NodeGet($head, ND_A);
    call ScanAddrTakenStmtList($body);
    goto SATSNext;
  end
 $tmp = $kind - NK_ASSIGN;
  if $tmp is zero:
   $tmp = call NodeGet($head, ND_B);
    call ScanAddrTakenExpr($tmp);
    goto SATSNext;
  end
 $tmp = $kind - NK_STORE;
  if $tmp is zero:
   $tmp = call NodeGet($head, ND_A);
    call ScanAddrTakenExpr($tmp);
   $tmp = call NodeGet($head, ND_B);
    call ScanAddrTakenExpr($tmp);
   $tmp = call NodeGet($head, ND_D);
    call ScanAddrTakenExpr($tmp);
```

```
    goto SATSNext;
  end
 $tmp = $kind - NK_CALL;
  if $tmp  is  zero:
   $tmp =  call NodeGet($head, ND_C);
    call ScanAddrTakenExprList($tmp);
    goto SATSNext;
  end
 $tmp = $kind - NK_CALLASN;
  if $tmp  is  zero:
   $tmp =  call NodeGet($head, ND_C);
    call ScanAddrTakenExprList($tmp);
    goto SATSNext;
  end
 $tmp = $kind - NK_IF;
  if $tmp  is  zero:
   $tmp =  call NodeGet($head, ND_A);
    call ScanAddrTakenExpr($tmp);
   $body =  call NodeGet($head, ND_B);
    call ScanAddrTakenStmtList($body);
   $elif =  call NodeGet($head, ND_C);
   SATSElifLoop:
    if $elif  is  zero:
      goto SATSElifDone;
    end
   $tmp =  call NodeGet($elif, ND_A);
    call ScanAddrTakenExpr($tmp);
   $tmp =  call NodeGet($elif, ND_B);
    call ScanAddrTakenStmtList($tmp);
   $elif =  call NodeGet($elif, ND_NEXT);
    goto SATSElifLoop;
   SATSElifDone:
   $tmp =  call NodeGet($head, ND_D);
    call ScanAddrTakenStmtList($tmp);
    goto SATSNext;
  end
 $tmp = $kind - NK_WHILE;
  if $tmp  is  zero:
   $tmp =  call NodeGet($head, ND_A);
    call ScanAddrTakenExpr($tmp);
   $tmp =  call NodeGet($head, ND_B);
    call ScanAddrTakenStmtList($tmp);
    goto SATSNext;
  end
 $tmp = $kind - NK_LOOP;
  if $tmp  is  zero:
   $tmp =  call NodeGet($head, ND_A);
    call ScanAddrTakenStmtList($tmp);
    goto SATSNext;
  end
 $tmp = $kind - NK_DO;
  if $tmp  is  zero:
   $tmp =  call NodeGet($head, ND_A);
    call ScanAddrTakenStmtList($tmp);
   $tmp =  call NodeGet($head, ND_B);
    call ScanAddrTakenExpr($tmp);
    goto SATSNext;
  end
 $tmp = $kind - NK_RETURN;
  if $tmp  is  zero:
   $tmp =  call NodeGet($head, ND_A);
    call ScanAddrTakenExprList($tmp);
```

```
    $tmp = call NodeGet($head, ND_C);
     call ScanAddrTakenExpr($tmp);
     goto SATSNext;
   end
  $tmp = $kind - NK_GOTO;
   if $tmp is zero:
    $tmp = call NodeGet($head, ND_A);
     call ScanAddrTakenExpr($tmp);
    $tmp = call NodeGet($head, ND_C);
     call ScanAddrTakenExpr($tmp);
     goto SATSNext;
   end
  $tmp = $kind - NK_TRAP;
   if $tmp is zero:
    $tmp = call NodeGet($head, ND_A);
     call ScanAddrTakenExpr($tmp);
    $tmp = call NodeGet($head, ND_B);
     call ScanAddrTakenExpr($tmp);
    $tmp = call NodeGet($head, ND_C);
     call ScanAddrTakenExpr($tmp);
     goto SATSNext;
   end
  $tmp = $kind - NK_WYDESET;
   if $tmp is zero:
    $tmp = call NodeGet($head, ND_B);
     call ScanAddrTakenExpr($tmp);
     goto SATSNext;
   end
  SATSNext:
  $head = call NodeGet($head, ND_NEXT);
   goto SATSLoop;
 end
```

⟨ ***Generator procedures*** ⟩ ≡

```
 def ScanAddrTakenTop($head):
   register $kind, $tmp, $body, $items;
  SATTopLoop:
   if $head is zero:


     return;
   end
  $kind = call NodeGet($head, ND_KIND);
  $tmp = $kind - NK_GREG;
   if $tmp is zero:
    $tmp = call NodeGet($head, ND_C);
     call ScanAddrTakenExpr($tmp);
    $tmp = call NodeGet($head, ND_D);
     call ScanAddrTakenExpr($tmp);
     goto SATTopNext;
   end
  $tmp = $kind - NK_CONST;
   if $tmp is zero:
    $tmp = call NodeGet($head, ND_C);
     call ScanAddrTakenExpr($tmp);
     goto SATTopNext;
   end
  $tmp = $kind - NK_DATADIR;
   if $tmp is zero:
```

```
    $items = call NodeGet($head, ND_D);
     call ScanAddrTakenExprList($items);
     goto SATTopNext;
   end
  $tmp = $kind - NK_LOC;
   if $tmp is zero:
    $tmp = call NodeGet($head, ND_A);
     call ScanAddrTakenExpr($tmp);
     goto SATTopNext;
   end
  $tmp = $kind - NK_RESERVE;
   if $tmp is zero:
    $tmp = call NodeGet($head, ND_C);
     call ScanAddrTakenExpr($tmp);
     goto SATTopNext;
   end
  $tmp = $kind - NK_DEF;
   if $tmp is zero:
    $body = call NodeGet($head, ND_D);
     call ScanAddrTakenStmtList($body);
     goto SATTopNext;
   end
   call ScanAddrTakenStmt($head);
   goto SATTopNext;
 SATTopNext:
  $head = call NodeGet($head, ND_NEXT);
   goto SATTopLoop;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def AliasEnterProc():
 $LocalAliasCnt = 0;
 $gen_in_proc = 1;
  call ClearRegUsed($LocalRegUsed);
  call MarkRegUsed($LocalRegUsed, 17);
  return;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def AliasLeaveProc():
 $LocalAliasCnt = 0;
 $gen_in_proc = 0;
  return;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def GenRegName($off, $len):
  register $ptr, $ch;
 $ptr = $StringPool + $off;
  if $len is zero:
    return;
  end
 $ch = [$ptr].bu;
 $ch = $ch - '$';
  if likely $ch is nonzero:
    call AppendBytes($ptr, $len);
    return;
  end
 $ptr = $ptr + 1;
 $len = $len - 1;
  if likely $len is nonzero:
    call AppendBytes($ptr, $len);
  end
  return;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def LookupAliasIn($base, $count, $off, $len) -> $r:
  register $idx, $ent, $eoff, $elen, $tmp, $reg, $p1, $p2;
 $r = 0;
 $idx = 0;
 LAILoop:
 $tmp = $idx - $count;
  goto LAIDone if $tmp is nonneg;
 $ent = $idx * ALIAS_ENTRY_SIZE;
 $ent = $ent + $base;
 $elen = [$ent + ALIAS_NAME_LEN].o;
 $tmp = $elen - $len;
  goto LAINext if $tmp is nonzero;
 $eoff = [$ent + ALIAS_NAME_OFF].o;
 $p1 = $StringPool + $eoff;
 $p2 = $StringPool + $off;
 $tmp = call StrEq($p1, $elen, $p2, $len);
  goto LAIHit if $tmp is nonzero;
 LAINext:
 $idx = $idx + 1;
  goto LAILoop;
 LAIHit:
 $reg = [$ent + ALIAS_REG_NUM].o;
 $r = $reg + 1;
 LAIDone:
  return $r;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def LookupAlias($off, $len) -> $r:
  if likely $gen_in_proc is nonzero:
   $r = call LookupAliasIn($LocalAliasBuf, $LocalAliasCnt, $off, $len);
    if $r is nonzero:
      return $r;
    end
  end
  return call LookupAliasIn($GlobalAliasBuf, $GlobalAliasCnt, $off, $len);
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def MarkRegUsed($base, $reg):
 [$base + $reg].bu = 1;
  return;
end
```

`MaxRegUsed` returns the highest register index marked used in the supplied bitmap, or -1 if none are set. Used by `GenCall` to size `$callbase` above any caller-frame local so that `PUSHJ`'s register-stack discipline does not clobber the caller's locals.

⟨ ***Generator procedures*** ⟩ ≡

```
def MaxRegUsed($used) -> $r:
  register $idx, $b, $oct;
  # The bitmap is octa-aligned (see the |$GlobalRegUsed| declaration
  # in the data segment), so we may load eight bytes at a time as a


  # single octa.  Each octa covers register indices |$idx..$idx+7|;
  # once a non-zero octa is found we drop into a byte-precise scan
  # over that octa to recover the highest set index.
 $idx = 248;
 MRUOctaLoop:
  goto MRUNone  if $idx  is  neg;
 $oct = [$used + $idx].o;
  goto MRUOctaHit  if $oct  is  nonzero;
 $idx = $idx - 8;
  goto MRUOctaLoop;
 MRUOctaHit:
 $idx = $idx + 7;
 MRUByteLoop:
 $b = [$used + $idx].bu;
  goto MRUFound  if $b  is  nonzero;
 $idx = $idx - 1;
  goto MRUByteLoop;
 MRUFound:
 $r = $idx;
  return $r;
 MRUNone:
 $r = -1;
  return $r;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def AllocFreeReg($used, $downward) -> $r:
  register $idx, $tmp;
  if $downward is zero:
   $idx = 0;

   AFRUpLoop:
   $tmp = $idx - 256;
    goto AFRFail if $tmp is nonneg;
   $tmp = [$used + $idx].bu;
    goto AFRUpHit if $tmp is zero;
   $idx = $idx + 1;
    goto AFRUpLoop;
   AFRUpHit:
   $r = $idx + 1;
    return $r;
  end
 $idx = 255;
 AFRDnLoop:
 $tmp = $idx + 1;
  goto AFRFail if $tmp is zero;
 $tmp = [$used + $idx].bu;
  goto AFRDnHit if $tmp is zero;
 $idx = $idx - 1;
  goto AFRDnLoop;
 AFRDnHit:
 $r = $idx + 1;
  return $r;
 AFRFail:
  call ErrHalt(&ErrAliasFull);
  return 0;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def BindAlias($local, $off, $len, $reg):
  register $base, $count, $ent, $tmp, $used;
  if $local is zero:
   $base = $GlobalAliasBuf;

   $count = $GlobalAliasCnt;
   $used = $GlobalRegUsed;
    goto BADo;
  end
 $base = $LocalAliasBuf;
 $count = $LocalAliasCnt;
 $used = $LocalRegUsed;
 BADo:
 $tmp = $count - ALIAS_MAX;
  if $tmp is nonneg:
    call ErrHalt(&ErrAliasFull);
  end
 $ent = $count * ALIAS_ENTRY_SIZE;
 $ent = $ent + $base;
 [$ent + ALIAS_NAME_OFF].o = $off;
 [$ent + ALIAS_NAME_LEN].o = $len;
 [$ent + ALIAS_REG_NUM].o = $reg;
  call MarkRegUsed($used, $reg);
```

```
  if $local is zero:
   $GlobalAliasCnt = $count + 1;
    return;
  end
 $LocalAliasCnt = $count + 1;
  return;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def ResolveAliasTarget($node) -> $r:
  register $kind, $tmp, $off, $len, $reg;
 $r = 0;
 $kind = call NodeGet($node, ND_KIND);
 $tmp = $kind - NK_REG;
  goto RATReg if $tmp is zero;
 $tmp = $kind - NK_REGNAME;
  goto RATName if $tmp is zero;
  return $r;
 RATReg:
 $reg = call NodeGet($node, ND_A);
 $r = $reg + 1;
  return $r;
 RATName:
 $off = call NodeGet($node, ND_A);
 $len = call NodeGet($node, ND_B);
  return call LookupAlias($off, $len);
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def EmitAliasDef($off, $len, $reg):
  register $tmp;
  call GenRegName($off, $len);
 $tmp = &StrIsDir;
  call AppendCStr($tmp);
  call AppendByte('$');
  call AppendInt($reg);
  call AppendByte('\n');
  return;
end
```

⟨ ***Generator procedures*** ⟩ ≡

```
def BindParamAliases($head, $slot) -> $r:
  register $off, $len, $reg, $flag;
 $r = $slot;
 BPLoop:
  if $head is zero:
    return $r;
  end
 $flag = call NodeGet($head, ND_D);
  if $flag is zero:
   $reg = $r;
    goto BPHaveReg;
  end
```

```
  $reg =  call NodeGet($head, ND_C);
  BPHaveReg:
  $off =  call NodeGet($head, ND_A);
  $len =  call NodeGet($head, ND_B);
   if $len  is  nonzero:
     call BindAlias(1, $off, $len, $reg);
   end
   call MarkRegUsed($LocalRegUsed, $reg);
  $r = $r + 1;
  $head =  call NodeGet($head, ND_NEXT);
   goto BPLoop;
end
```

24. Driver banner.

The standalone compiler driver wraps every compilation in a two-line MMIXAL comment identifying the source file and warning that the output is machine-generated. `GenBanner` is the self-hosted analogue: it is invoked once at the top of `Generate` and emits the same two lines, followed by a blank line, using the `SourceNameBuf` filled in by `InitSourceName` at start-up. The banner lives in the generator rather than in `main` so the output stream goes through the same `OutputBuf`/`AppendCStr` discipline as everything else, and so tangled tests can diff the whole file including the banner.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenBanner():
   register $tmp, $p, $ch;
  $tmp = &StrBannerA;
   call AppendCStr($tmp);
  $p = $SourceNameBuf;
  GBnLp:
  $ch = [$p].bu;
   goto GBnDn  if $ch  is  zero;
   call AppendByte($ch);
  $p = $p + 1;
   goto GBnLp;
  GBnDn:
  $tmp = &StrBannerB;
   call AppendCStr($tmp);
   return;
end
```

⟨ ***Data segment*** ⟩ +≡

```
byte StrBannerA   "% Generated by plmc from ", 0;
byte StrBannerB   "\n% DO NOT EDIT -- edit the source file instead.\n\n", 0;
```

`GenSegCode` emits the standard code-segment preamble `LOC \#100\\nMain IS @\\n` the first time a `segment code;` directive is reached in the top-level walk, and degrades to a bare `LOC #100` on any subsequent appearance (a well-formed program will only have one, but the duplicate-safe fallback keeps tangled intermediate output syntactically valid).

⟨ ***Generator procedures*** ⟩ +≡

```
def GenSegCode($node):
  register $tmp;
 $tmp = &StrSegCode;
  call AppendCStr($tmp);
 $tmp = $main_emitted - 1;
  if $tmp  is  zero:
    return;
  end
 $tmp = &StrMainIs;
  call AppendCStr($tmp);
 $main_emitted = 1;
  return;
end
```

GenTopItem dispatches on the ND_KIND field of a top-level AST node. Each recognized top-level form is forwarded to the dedicated emitter for register declarations, prefixes, segments, constants, storage directives, and procedure bodies; any executable statement that appears at top level falls through to GenStatement.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenTopItem($node):
  register $kind, $tmp;
 $kind =  call NodeGet($node, ND_KIND);
 $tmp = $kind - NK_REGDECL;


  goto GTIRgd   if $tmp  is  zero;
 $tmp = $kind - NK_PREFIX;
  goto GTIPref  if $tmp  is  zero;
 $tmp = $kind - NK_POOL;
  goto GTIPool  if $tmp  is  zero;
 $tmp = $kind - NK_LOCAL;
  goto GTILoc   if $tmp  is  zero;
 $tmp = $kind - NK_SEGDATA;
  goto GTISegD  if $tmp  is  zero;
 $tmp = $kind - NK_SEGCODE;
  goto GTISegC  if $tmp  is  zero;
 $tmp = $kind - NK_LOC;
  goto GTILc    if $tmp  is  zero;
 $tmp = $kind - NK_CONST;
  goto GTICon   if $tmp  is  zero;
 $tmp = $kind - NK_RESERVE;
  goto GTIRes   if $tmp  is  zero;
 $tmp = $kind - NK_GREG;
  goto GTIGrg   if $tmp  is  zero;
 $tmp = $kind - NK_DATADIR;
  goto GTIDat   if $tmp  is  zero;
 $tmp = $kind - NK_DEF;
  goto GTIDef   if $tmp  is  zero;
  call GenStatement($node);
  return;
 GTIRgd:
  call GenRegList($node);
  return;
 GTIPref:
  call GenPrefix($node);
  return;
```

```
    GTIPool:
    $tmp = &StrPool;
     call AppendCStr($tmp);
     return;
    GTILoc:
     call GenLocal($node);
     return;
    GTISegD:
     call GenSegData($node);
     return;
    GTISegC:
     call GenSegCode($node);
     return;
    GTILc:
     call GenLoc($node);
     return;
    GTICon:
     call GenConst($node);
     return;
    GTIRes:
     call GenReserve($node);
     return;
    GTIGrg:
     call GenGreg($node);
     return;
    GTIDat:
     call GenDataDir($node);
     return;
    GTIDef:
     call GenDef($node);
     return;
   end
```

`GenLocal` emits `LOCAL $N` for a `local $N;` declaration. The register number is stored in `ND_A`.

⟨ ***Generator procedures*** ⟩ +≡

```
   def GenLocal($node):
     register $num, $tmp;
    $tmp = &StrLocal;
     call AppendCStr($tmp);
    $num =  call NodeGet($node, ND_A);
     call AppendByte('$');
     call AppendInt($num);
     call AppendByte('\n');
     return;
   end
```

`GenPrefix` emits `PREFIX name:` (or the bare `PREFIX :` form, which clears any pending prefix) for an `NK_PREFIX` top-level node. The node's `ND_A`/`ND_B` fields are a string-pool offset / length pair naming the prefix; an `ND_A` of zero indicates the bare form.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenPrefix($node):
  register $off, $len, $ptr, $tmp;
 $tmp = &StrPrefix;
  call AppendCStr($tmp);
 $off =  call NodeGet($node, ND_A);
 $len =  call NodeGet($node, ND_B);
  if $len  is  zero:
    call AppendByte(':');
    call AppendByte('\n');
    return;
  end
 $ptr = $StringPool + $off;
  call AppendBytes($ptr, $len);
  call AppendByte(':');
  call AppendByte('\n');
  return;
end
```

`GenLabelName` writes a `TK_LABEL`-style name (pool-offset / length pair stored in `ND_A`/`ND_B` of the argument node) to `OutputBuf`. Used by emitters for any top-level directive whose node begins with a label name.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenLabelName($node):
  register $off, $len, $ptr;
 $off =  call NodeGet($node, ND_A);
 $len =  call NodeGet($node, ND_B);
 $ptr = $StringPool + $off;
  call AppendBytes($ptr, $len);
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def GenScopedLabelName($node):
  register $off, $len;
 $off =  call NodeGet($node, ND_A);
 $len =  call NodeGet($node, ND_B);
  if  likely $gen_in_proc  is  nonzero:
    if  likely $ProcNameLen  is  nonzero:
      call GenPoolName($ProcNameOff, $ProcNameLen);
      call AppendByte('_');
      call AppendByte('_');
    end
  end
  call GenPoolName($off, $len);
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def GenScopedLabelPoolName($off, $len):
  if likely $gen_in_proc is nonzero:
    if likely $ProcNameLen is nonzero:
      call GenPoolName($ProcNameOff, $ProcNameLen);
      call AppendByte('_');
      call AppendByte('_');
    end
  end
  call GenPoolName($off, $len);
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def ProcLabelReset():
 $ProcLabelCnt = 0;
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def DupLabelBump($off, $len):
  register $ent, $cnt;
 $ent = call SymIntern($off, $len);
 $cnt = [$ent + SYM_DUP_COUNT].o;
 $cnt = $cnt + 1;
 [$ent + SYM_DUP_COUNT].o = $cnt;
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def IsDupLabel($off, $len) -> $r:
  register $ent, $cnt, $tmp;
 $r = 0;
 $ent = call SymLookup($off, $len);
  if $ent is zero:
    return $r;
  end
 $cnt = [$ent + SYM_DUP_COUNT].o;
 $tmp = $cnt - 2;
  if $tmp is nonneg:
   $r = 1;
  end
  return $r;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def RegisterProcLabel($off, $len):
  register $ent, $tmp;
 $tmp = $ProcLabelCnt - PROC_LABEL_MAX;
  if $tmp  is  nonneg:
    call ErrHalt(&ErrAliasFull);
  end
 $ent = $ProcLabelCnt << 4;          # * PROC_LABEL_ENTRY_SIZE (= 16)
 $ent = $ent + $ProcLabelBuf;
 [$ent + PROC_LABEL_OFF].o = $off;
 [$ent + PROC_LABEL_LEN].o = $len;
 $ProcLabelCnt = $ProcLabelCnt + 1;
  return;
end
```

`ProcLabelMode` classifies a label name relative to the currently-emitting procedure. Result 0 means "not a label of this proc" (so emit unmangled), 1 means "proc-local label whose name appears only in this proc" (so emit unmangled), 2 means "proc-local label whose name also appears in another proc" (so prefix with the proc name to disambiguate against MMIXAL's flat namespace). Membership comes from `$ProcLabelBuf`; the duplicate flag is read from the global symbol table via `IsDupLabel`.

⟨ ***Generator procedures*** ⟩ +≡

```
def ProcLabelMode($off, $len) -> $r:
  register $idx, $ent, $eoff, $elen, $tmp, $p1, $p2;
 $r = 0;
 $idx = 0;
 IPLLoop:
 $tmp = $idx - $ProcLabelCnt;
  goto IPLDone  if $tmp  is  nonneg;
 $ent = $idx << 4;                   # * PROC_LABEL_ENTRY_SIZE (= 16)
 $ent = $ent + $ProcLabelBuf;
 $elen = [$ent + PROC_LABEL_LEN].o;
 $tmp = $elen - $len;
  goto IPLNext  if $tmp  is  nonzero;
 $eoff = [$ent + PROC_LABEL_OFF].o;
 $p1 = $StringPool + $eoff;
 $p2 = $StringPool + $off;
 $tmp =  call StrEq($p1, $elen, $p2, $len);
  goto IPLHit  if $tmp  is  nonzero;
 IPLNext:
 $idx = $idx + 1;
  goto IPLLoop;
 IPLHit:
 $tmp =  call IsDupLabel($off, $len);
 $r = $tmp + 1;
 IPLDone:
  return $r;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def CollectProcLabelsOne($node):
  register $kind, $tmp, $off, $len, $head, $elif;
  if $node is zero:
    return;


  end
 $kind = call NodeGet($node, ND_KIND);
 $tmp = $kind - NK_LABELSTMT;
  if $tmp is zero:
   $off = call NodeGet($node, ND_A);
   $len = call NodeGet($node, ND_B);
    call RegisterProcLabel($off, $len);
    return;
  end
 $tmp = $kind - NK_BEGIN;
  if $tmp is zero:
   $head = call NodeGet($node, ND_A);
    call CollectProcLabels($head);
    return;
  end
 $tmp = $kind - NK_WHILE;
  if $tmp is zero:
   $head = call NodeGet($node, ND_B);
    call CollectProcLabels($head);
    return;
  end
 $tmp = $kind - NK_LOOP;
  if $tmp is zero:
   $head = call NodeGet($node, ND_A);
    call CollectProcLabels($head);
    return;
  end
 $tmp = $kind - NK_DO;
  if $tmp is zero:
   $head = call NodeGet($node, ND_A);
    call CollectProcLabels($head);
    return;
  end
 $tmp = $kind - NK_IF;
  if $tmp is nonzero:
    return;
  end
 $head = call NodeGet($node, ND_B);
  call CollectProcLabels($head);
 $elif = call NodeGet($node, ND_C);
 CPLIfElif:
  if $elif is zero:
    goto CPLIfElse;
  end
 $head = call NodeGet($elif, ND_B);
  call CollectProcLabels($head);
 $elif = call NodeGet($elif, ND_NEXT);
  goto CPLIfElif;
 CPLIfElse:
 $head = call NodeGet($node, ND_D);
  call CollectProcLabels($head);
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def CollectProcLabels($head):
 CPLLoop:
  if $head is zero:
    return;
  end
  call CollectProcLabelsOne($head);
 $head = call NodeGet($head, ND_NEXT);
  goto CPLLoop;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def AnnotateProcExpr($node):
  register $kind, $tmp, $off, $len, $lhs, $rhs;
  if $node is zero:
    return;


  end
 $kind = call NodeGet($node, ND_KIND);
 $tmp = $kind - NK_GETA;
  if $tmp is zero:
   $off = call NodeGet($node, ND_A);
   $len = call NodeGet($node, ND_B);
   $tmp = call ProcLabelMode($off, $len);
    if $tmp is zero:
      return;
    end
   $rhs = call IsDeclaredDataLabelName($off, $len);
    if $rhs is zero:
      call NodeSet($node, ND_C, $tmp);
    end
    return;
  end
 $tmp = $kind - NK_UNOP;
  if $tmp is zero:
   $lhs = call NodeGet($node, ND_A);
    call AnnotateProcExpr($lhs);
    return;
  end
 $tmp = $kind - NK_BINOP;
  if $tmp is zero:
   $lhs = call NodeGet($node, ND_A);
    call AnnotateProcExpr($lhs);
   $rhs = call NodeGet($node, ND_B);
    call AnnotateProcExpr($rhs);
    return;
  end
 $tmp = $kind - NK_LOAD;
  if $tmp is zero:
   $lhs = call NodeGet($node, ND_A);
    call AnnotateProcExpr($lhs);
   $rhs = call NodeGet($node, ND_B);
    call AnnotateProcExpr($rhs);
    return;
  end
 $tmp = $kind - NK_COND;
  if $tmp is nonzero:
    return;
```

```
    end
  $rhs =  call NodeGet($node, ND_D);
   if $rhs  is  nonzero:
     call AnnotateProcExpr($rhs);
   end
   return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def AnnotateProcExprList($head):
 APELLoop:
  if $head  is  zero:
    return;
  end
  call AnnotateProcExpr($head);
 $head =  call NodeGet($head, ND_NEXT);
  goto APELLoop;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def AnnotateProcStmtOne($node):
  register $kind, $tmp, $off, $len, $head, $elif, $cond;
  if $node  is  zero:
    return;


  end
 $kind =  call NodeGet($node, ND_KIND);
 $tmp = $kind - NK_LABELSTMT;
  if $tmp  is  zero:
   $off =  call NodeGet($node, ND_A);
   $len =  call NodeGet($node, ND_B);
   $tmp =  call ProcLabelMode($off, $len);
   $tmp = $tmp - 2;
    if $tmp  is  zero:
      call NodeSet($node, ND_C, 1);
    end
    return;
  end
 $tmp = $kind - NK_GOTO;
  if $tmp  is  zero:
   $cond =  call NodeGet($node, ND_C);
    call AnnotateProcExpr($cond);
    return;
  end
 $tmp = $kind - NK_ASSIGN;
  if $tmp  is  zero:
   $head =  call NodeGet($node, ND_B);
    call AnnotateProcExpr($head);
    return;
  end
 $tmp = $kind - NK_STORE;
  if $tmp  is  zero:
   $head =  call NodeGet($node, ND_A);
    call AnnotateProcExpr($head);
   $head =  call NodeGet($node, ND_B);
```

```
    call AnnotateProcExpr($head);
   $head =  call NodeGet($node, ND_D);
    call AnnotateProcExpr($head);
    return;
  end
 $tmp = $kind - NK_CALL;
  if $tmp  is  zero:
   $head =  call NodeGet($node, ND_C);
    call AnnotateProcExprList($head);
   $tmp =  call NodeGet($node, ND_D);
    if $tmp  is  nonzero:
     $head =  call NodeGet($node, ND_B);
      call AnnotateProcExpr($head);
    end
    return;
  end
 $tmp = $kind - NK_CALLASN;
  if $tmp  is  zero:
   $head =  call NodeGet($node, ND_C);
    call AnnotateProcExprList($head);
   $tmp =  call NodeGet($node, ND_D);
    if $tmp  is  nonzero:
     $head =  call NodeGet($node, ND_B);
      call AnnotateProcExpr($head);
    end
    return;
  end
 $tmp = $kind - NK_BEGIN;
  if $tmp  is  zero:
   $head =  call NodeGet($node, ND_A);
    call AnnotateProcStmtList($head);
    return;
  end
 $tmp = $kind - NK_WHILE;
  if $tmp  is  zero:
   $cond =  call NodeGet($node, ND_A);
    call AnnotateProcExpr($cond);
   $head =  call NodeGet($node, ND_B);
    call AnnotateProcStmtList($head);
    return;
  end
 $tmp = $kind - NK_LOOP;
  if $tmp  is  zero:
   $head =  call NodeGet($node, ND_A);
    call AnnotateProcStmtList($head);
    return;
  end
 $tmp = $kind - NK_DO;
  if $tmp  is  zero:
   $head =  call NodeGet($node, ND_A);
    call AnnotateProcStmtList($head);
   $cond =  call NodeGet($node, ND_B);
    call AnnotateProcExpr($cond);
    return;
  end
 $tmp = $kind - NK_IF;
  if $tmp  is  zero:
   $cond =  call NodeGet($node, ND_A);
    call AnnotateProcExpr($cond);
   $head =  call NodeGet($node, ND_B);
    call AnnotateProcStmtList($head);
   $elif =  call NodeGet($node, ND_C);
```

```
    APSIfElif:
     if $elif is zero:
       goto APSIfElse;
     end
    $cond = call NodeGet($elif, ND_A);
     call AnnotateProcExpr($cond);
    $head = call NodeGet($elif, ND_B);
     call AnnotateProcStmtList($head);
    $elif = call NodeGet($elif, ND_NEXT);
     goto APSIfElif;
    APSIfElse:
    $head = call NodeGet($node, ND_D);
     call AnnotateProcStmtList($head);
     return;
   end
  $tmp = $kind - NK_RETURN;
   if $tmp is zero:
    $cond = call NodeGet($node, ND_C);
     call AnnotateProcExpr($cond);
     return;
   end
   return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def AnnotateProcStmtList($head):
 APSLoop:
  if $head is zero:
    return;
  end
  call AnnotateProcStmtOne($head);
 $head = call NodeGet($head, ND_NEXT);
  goto APSLoop;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def ScanDupLabels($head):
  register $kind, $tmp, $body, $idx, $ent, $off, $len;
 SDLLoop:
  if $head is zero:
    return;
  end
 $kind = call NodeGet($head, ND_KIND);
 $tmp = $kind - NK_DEF;
  goto SDLNext if $tmp is nonzero;
 $body = call NodeGet($head, ND_D);
  call ProcLabelReset();
  call CollectProcLabels($body);
 $idx = 0;
 SDLBump:
 $tmp = $idx - $ProcLabelCnt;
  goto SDLNext if $tmp is nonneg;
 $ent = $idx << 4;                    # * PROC_LABEL_ENTRY_SIZE (= 16)
 $ent = $ent + $ProcLabelBuf;
 $off = [$ent + PROC_LABEL_OFF].o;
 $len = [$ent + PROC_LABEL_LEN].o;
```

```
    call DupLabelBump($off, $len);
  $idx = $idx + 1;
    goto SDLBump;
  SDLNext:
  $head =  call NodeGet($head, ND_NEXT);
    goto SDLLoop;
  end
```

`GenPoolName` writes a pool-offset / length name pair directly given the two integers. Used by expression nodes that hold a name in `ND_A`/`ND_B`.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenPoolName($off, $len):
  register $ptr;
 $ptr = $StringPool + $off;
  call AppendBytes($ptr, $len);
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def LookupBuiltinImm($off, $len) -> $r:
  register $ptr, $tmp, $sym, $symlen;
 $ptr = $StringPool + $off;


 $sym = &StrSymHALT;
 $symlen = 4;
 $tmp =  call StrEq($ptr, $len, $sym, $symlen);
  if $tmp  is  nonzero:
    return 1;
  end

 $sym = &StrSymFOPEN;
 $symlen = 5;
 $tmp =  call StrEq($ptr, $len, $sym, $symlen);
  if $tmp  is  nonzero:
    return 2;
  end

 $sym = &StrSymFCLOSE;
 $symlen = 6;
 $tmp =  call StrEq($ptr, $len, $sym, $symlen);
  if $tmp  is  nonzero:
    return 3;
  end

 $sym = &StrSymFREAD;
 $symlen = 5;
 $tmp =  call StrEq($ptr, $len, $sym, $symlen);
  if $tmp  is  nonzero:
    return 4;
  end

 $sym = &StrSymFGETS;
 $symlen = 5;
```

```
$tmp =  call StrEq($ptr, $len, $sym, $symlen);
 if $tmp  is  nonzero:
   return 5;
 end

$sym = &StrSymFGETWS;
$symlen = 6;
$tmp =  call StrEq($ptr, $len, $sym, $symlen);
 if $tmp  is  nonzero:
   return 6;
 end

$sym = &StrSymFWRITE;
$symlen = 6;
$tmp =  call StrEq($ptr, $len, $sym, $symlen);
 if $tmp  is  nonzero:
   return 7;
 end

$sym = &StrSymFPUTS;
$symlen = 5;
$tmp =  call StrEq($ptr, $len, $sym, $symlen);
 if $tmp  is  nonzero:
   return 8;
 end

$sym = &StrSymFPUTWS;
$symlen = 6;
$tmp =  call StrEq($ptr, $len, $sym, $symlen);
 if $tmp  is  nonzero:
   return 9;
 end

$sym = &StrSymFSEEK;
$symlen = 5;
$tmp =  call StrEq($ptr, $len, $sym, $symlen);
 if $tmp  is  nonzero:
   return 10;
 end

$sym = &StrSymFTELL;
$symlen = 5;
$tmp =  call StrEq($ptr, $len, $sym, $symlen);
 if $tmp  is  nonzero:
   return 11;
 end

$sym = &StrSymSTDIN;
$symlen = 5;
$tmp =  call StrEq($ptr, $len, $sym, $symlen);
 if $tmp  is  nonzero:
   return 1;
 end

$sym = &StrSymSTDOUT;
$symlen = 6;
$tmp =  call StrEq($ptr, $len, $sym, $symlen);
 if $tmp  is  nonzero:
   return 2;
 end

$sym = &StrSymSTDERR;
```

```
  $symlen = 6;
  $tmp =  call StrEq($ptr, $len, $sym, $symlen);
   if $tmp  is  nonzero:
     return 3;
   end

   return 0;
 end
```

`GenConstExpr` emits a PL/MMIX constant expression as MMIXAL text. The grammar admits only `NK_IMM`, `NK_LABELREF`, `NK_UNOP` (with `TK_MINUS` as the op code), and `NK_BINOP` (with `TK_PLUS`, `TK_MINUS`, `TK_STAR`, or `TK_SLASH`). MMIXAL's expression grammar has the same left-to-right "flat" precedence used by the PL/MMIX const-expr parser, so we can emit lhs/op/rhs without parenthesising.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenConstExpr($node):
   register $kind, $tmp, $a, $b, $op, $val, $off, $len;
  $kind =  call NodeGet($node, ND_KIND);
  $tmp = $kind - NK_IMM;


   goto GCEImm   if $tmp  is  zero;
  $tmp = $kind - NK_FLOAT;
   goto GCEFlt   if $tmp  is  zero;
  $tmp = $kind - NK_LABELREF;
   goto GCELab   if $tmp  is  zero;
  $tmp = $kind - NK_UNOP;
   goto GCEUn    if $tmp  is  zero;
  $tmp = $kind - NK_BINOP;
   goto GCEBin   if $tmp  is  zero;
   # Fallback: emit "?" so bad input is visible but the stream stays
   # well-formed.
   call AppendByte('?');
   return;
  GCEImm:
  $val =  call NodeGet($node, ND_A);
   if $val  is  neg:
     call AppendHexOcta($val);
     return;
   end
   call AppendInt($val);
   return;
  GCEFlt:
  $off =  call NodeGet($node, ND_A);
  $len =  call NodeGet($node, ND_B);
  $tmp = $StringPool + $off;
   call AppendBytes($tmp, $len);
   return;
  GCELab:
  $off =  call NodeGet($node, ND_A);
  $len =  call NodeGet($node, ND_B);
  $tmp =  call LookupBuiltinImm($off, $len);
   if $tmp  is  nonzero:
    $tmp = $tmp - 1;
     call AppendInt($tmp);
     return;
   end
   call GenPoolName($off, $len);
```

```
    return;
  GCEUn:
  $op =  call NodeGet($node, ND_C);
   call GenOpChar($op);
  $a  =  call NodeGet($node, ND_A);
   call GenConstExpr($a);
   return;
  GCEBin:
  $a  =  call NodeGet($node, ND_A);
  $b  =  call NodeGet($node, ND_B);
  $op =  call NodeGet($node, ND_C);
   call GenConstExpr($a);
   call GenOpChar($op);
   call GenConstExpr($b);
   return;
end
```

`GenOpChar` writes the single-character MMIXAL spelling of an arithmetic operator token (`TK_PLUS`, `TK_MINUS`, `TK_STAR`, `TK_SLASH`). Any other code emits `?`.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenOpChar($op):
  register $tmp;
 $tmp = $op - TK_PLUS;
  if $tmp  is  zero:
    call AppendByte('+');
    return;
  end
 $tmp = $op - TK_MINUS;
  if $tmp  is  zero:
    call AppendByte('-');
    return;
  end
 $tmp = $op - TK_STAR;
  if $tmp  is  zero:
    call AppendByte('*');
    return;
  end
 $tmp = $op - TK_SLASH;
  if $tmp  is  zero:
    call AppendByte('/');
    return;
  end
  call AppendByte('?');
  return;
end
```

`GenSegData` emits the MMIXAL prologue for a segment-data directive with an optional base register. The unconditional `LOC Data_Segment` line positions the location counter; when the optional base-register name `$Base` was given (that is, `ND_A` is nonzero), we additionally emit a `$Base GREG @` line that binds the named global register to the current data-segment origin. The `$` prefix is part of the regname token's pool entry so we do not prepend it here.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenSegData($node):
  register $tmp, $off, $len;
 $tmp = &StrSegData;
  call AppendCStr($tmp);
 $off =  call NodeGet($node, ND_A);
 $len =  call NodeGet($node, ND_B);
  if $len  is  zero:
    return;
  end
  call RegisterGregName($off, $len);
  call GenRegName($off, $len);
 $tmp = &StrGregHere;
  call AppendCStr($tmp);
  return;
end
```

GenLoc emits `LOC <expr>\n` for an NK_LOC node. The address expression is in ND_A.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenLoc($node):
  register $tmp, $expr;
 $tmp = &StrLocDir;
  call AppendCStr($tmp);
 $expr =  call NodeGet($node, ND_A);
  call GenConstExpr($expr);
  call AppendByte('\n');
 $user_loc_seen = 1;
  return;
end
```

GenConst emits `name IS <expr>\n` for an NK_CONST node. The name is in ND_A/ND_B, the expression handle in ND_C.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenConst($node):
  register $tmp, $expr, $off, $len;
 $off =  call NodeGet($node, ND_A);
 $len =  call NodeGet($node, ND_B);
 $expr =  call NodeGet($node, ND_C);
  call RegisterDataLabel($off, $len);
  call RegisterConstDef($off, $len, $expr);
  call GenLabelName($node);
 $tmp = &StrIsDir;
  call AppendCStr($tmp);
  call GenConstExpr($expr);
  call AppendByte('\n');
  return;
end
```

GenReserve emits `name LOC @+<expr>\n` for an NK_RESERVE node with a name, or `LOC @+<expr>\n` for the anonymous form. The byte-count expression is in ND_C; the optional name is in ND_A/ND_B, with a zero ND_A signalling the anonymous form.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenReserve($node):
  register $tmp, $off, $expr, $len;
 $off =  call NodeGet($node, ND_A);
  if $off  is  zero:
   $tmp = &StrIndent8;
    call AppendCStr($tmp);
    goto GRsDir;
  end
 $len =  call NodeGet($node, ND_B);
 $tmp =  call IsDataLabelAddrTaken($off, $len);
  if $tmp  is  nonzero:
    call EmitDataGregName($off, $len);
   $tmp = &StrGregHere;
    call AppendCStr($tmp);
  end
  call GenLabelName($node);
 GRsDir:
 $tmp = &StrLocAtPlus;
  call AppendCStr($tmp);
 $expr =  call NodeGet($node, ND_C);
  call GenConstExpr($expr);
  call AppendByte('\n');
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def AppendDataDirOp($kw):
  register $tmp;
 $tmp = &StrIndent8;
  call AppendCStr($tmp);


 $tmp = $kw - KW_BYTE;
  goto ADDOByte  if $tmp  is  zero;
 $tmp = $kw - KW_WYDE;
  goto ADDOWyde  if $tmp  is  zero;
 $tmp = $kw - KW_TETRA;
  goto ADDOTetra  if $tmp  is  zero;
  goto ADDOOcta;
 ADDOByte:
  call AppendByte('B');
  call AppendByte('Y');
  call AppendByte('T');
  call AppendByte('E');
  call AppendByte(' ');
  call AppendByte(' ');
  call AppendByte(' ');
  call AppendByte(' ');
  return;
 ADDOWyde:
  call AppendByte('W');
  call AppendByte('Y');
  call AppendByte('D');
  call AppendByte('E');
  call AppendByte(' ');
  call AppendByte(' ');
  call AppendByte(' ');
  call AppendByte(' ');
```

```
    return;
  ADDOTetra:
    call AppendByte('T');
    call AppendByte('E');
    call AppendByte('T');
    call AppendByte('R');
    call AppendByte('A');
    call AppendByte(' ');
    call AppendByte(' ');
    call AppendByte(' ');
    return;
  ADDOOcta:
    call AppendByte('O');
    call AppendByte('C');
    call AppendByte('T');
    call AppendByte('A');
    call AppendByte(' ');
    call AppendByte(' ');
    call AppendByte(' ');
    call AppendByte(' ');
    return;
  end
```

GenGreg emits a `name GREG <init>\n` line for an NK_GREG node. The three parser-accepted forms are:

- • bare `greg $name;` — ND_C= 0, ND_D= 0, emits `GREG 0`.
- • `greg name[SIZE];` — ND_D≠ 0, emits `GREG @` (base-of-block), followed on a new line by `LOC @+SIZE` to reserve the block.
- • `greg name = EXPR;` — ND_C≠ 0, emits `GREG EXPR`.

The name, if present, is in ND_A/ND_B; a zero ND_A means an anonymous `GREG` declaration (used for filler slots).

⟨ ***Generator procedures*** ⟩ +≡

```
def GenGreg($node):
  register $tmp, $off, $len, $init, $size;
 $off =  call NodeGet($node, ND_A);
 $len =  call NodeGet($node, ND_B);


  if $len  is  nonzero:
    call RegisterGregName($off, $len);
  end
  if $len  is  zero:
   $tmp = &StrIndent8;
    call AppendCStr($tmp);
    goto GGrDir;
  end
  call GenRegName($off, $len);
 GGrDir:
 $tmp = &StrGregDir;
  call AppendCStr($tmp);
 $init =  call NodeGet($node, ND_C);
 $size =  call NodeGet($node, ND_D);
  if $init  is  nonzero:
   $tmp =  call GenGregFloatInit($init);
    if $tmp  is  nonzero:
```

```
      call AppendByte('\n');
      return;
    end
    call GenConstExpr($init);
    call AppendByte('\n');
    return;
  end
  if $size is nonzero:
    call AppendByte('@');
    call AppendByte('\n');
   $tmp = &StrLocAtPlus;
    call AppendCStr($tmp);
    call GenConstExpr($size);
    call AppendByte('\n');
    return;
  end
  call AppendByte('@');
  call AppendByte('\n');
  return;
end
```

`GenGregFloatInit` handles the one float-initializer family currently needed by the example suite: decimal literals whose fractional part is all zeros, such as `5.0` or `400.0`. Those are converted into their exact IEEE-754 octabyte bit pattern with `FLOT` so the emitted `GREG` line uses the canonical `#hhhh...` octabyte form. Any other float shape is left to the ordinary constant-expression emitter.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenGregFloatInit($node) -> $r:
  register $kind, $off, $len, $ptr, $end, $ch, $tmp, $ival, $saw_digit, $frac_len, $bits;
 $r = 0;
 $kind = call NodeGet($node, ND_KIND);


 $tmp = $kind - NK_FLOAT;
 if $tmp is nonzero:
   return $r;
 end
 $off = call NodeGet($node, ND_A);
 $len = call NodeGet($node, ND_B);
 $ptr = $StringPool + $off;
 $end = $ptr + $len;
 $ival = 0;
 $saw_digit = 0;
 GGFILoop:
 if $ptr >=u $end:
   return $r;
 end
 $ch = [$ptr].bu;
 $tmp = $ch - '.';
 goto GGFIDot if $tmp is zero;
 $tmp = $ch - '0';
 goto GGFINotHandled if $tmp is neg;
 $tmp = $ch - '9';
 goto GGFINotHandled if $tmp is pos;
 $ival = $ival * 10;
 $tmp = $ch - '0';
 $ival = $ival + $tmp;
 $saw_digit = 1;
```

```
  $ptr = $ptr + 1;
   goto GGFILoop;
  GGFIDot:
   goto GGFINotHandled  if $saw_digit  is  zero;
  $ptr = $ptr + 1;
  $frac_len = 0;
  GGFIFrac:
   if $ptr >=u $end:
     goto GGFIDone;
   end
  $ch = [$ptr].bu;
  $tmp = $ch - '0';
   goto GGFINotHandled  if $tmp  is  nonzero;
  $frac_len = $frac_len + 1;
  $ptr = $ptr + 1;
   goto GGFIFrac;
  GGFIDone:
   goto GGFINotHandled  if $frac_len  is  zero;
  $bits = $ival;
  $bits = flot. $bits;
   call AppendHexOcta($bits);
   return 1;
  GGFINotHandled:
   return $r;
end
```

GenDataDir emits `name <DIR> item, item, ...\n` for an NK_DATADIR node (`byte`, `wyde`, `tetra`, or `octa`). The keyword code in ND_C is mapped to the matching MMIXAL pseudo-op; the name is in ND_A/ND_B; the item-list head is in ND_D, linked through ND_NEXT. Each item is emitted through GenDataItem, which handles string, integer, negated, and label leaves.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenDataDir($node):
  register $tmp, $kw, $head, $off, $len, $alias, $greg;
 $off =  call NodeGet($node, ND_A);
 $len =  call NodeGet($node, ND_B);


 $alias = 0;
 $greg = 0;
  if $len  is  nonzero:
   $greg =  call IsGregName($off, $len);
    if $greg  is  zero:
     $alias =  call LookupAlias($off, $len);
    end
    if $alias  is  zero:
      if $greg  is  zero:
      call RegisterDataLabel($off, $len);
      end
    end
  end
  if $len  is  zero:
   $tmp = &StrIndent8;
    call AppendCStr($tmp);
  end
  if $len  is  nonzero:
    if $alias  is  zero:
      if $greg  is  zero:
       $tmp =  call IsDataLabelAddrTaken($off, $len);
```

```
        if $tmp is nonzero:
          call EmitDataGregName($off, $len);
         $tmp = &StrGregHere;
          call AppendCStr($tmp);
        end
        call GenLabelName($node);
      end
    end
    if $alias is nonzero:
     $tmp = &StrIndent8;
      call AppendCStr($tmp);
    end
    if $greg is nonzero:
     $tmp = &StrIndent8;
      call AppendCStr($tmp);
    end
  end
 $kw = call NodeGet($node, ND_C);
  call AppendDataDirOp($kw);
 $head = call NodeGet($node, ND_D);
 GDDLoop:
  if $head is zero:
    call AppendByte('\n');
    return;
  end
  call GenDataItem($head, $kw);
 $head = call NodeGet($head, ND_NEXT);
  if $head is zero:
    call AppendByte('\n');
    return;
  end
  call AppendByte('\n');
 $tmp = &StrIndent8;
  call AppendCStr($tmp);
  call AppendDataDirOp($kw);
  goto GDDLoop;
end
```

`GenDataItem` emits one data-literal leaf. Supported leaf kinds are `NK_STRING` (pool-held text rendered as a quoted string), `NK_IMM`, `NK_LABELREF`, and `NK_UNOP` (negated atom). Anything else emits a `?` placeholder so malformed input is visible without breaking the surrounding line.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenDataItem($node, $kw):
  register $kind, $tmp, $val, $off, $len, $inner, $ptr, $i, $ch, $open, $first, $chunk, $cols;
 $kind = call NodeGet($node, ND_KIND);
 $tmp = $kind - NK_STRING;


  goto GDIStr   if $tmp is zero;
 $tmp = $kind - NK_IMM;
  goto GDIImm   if $tmp is zero;
 $tmp = $kind - NK_LABELREF;
  goto GDILab   if $tmp is zero;
 $tmp = $kind - NK_UNOP;
  goto GDINeg   if $tmp is zero;
  call AppendByte('?');
  return;
 GDIStr:
```

```
$off =  call NodeGet($node, ND_A);
$len =  call NodeGet($node, ND_B);
$ptr = $StringPool + $off;
$i = 0;
$open = 0;
$first = 1;
$chunk = 0;
$cols = 0;
GDIStrLp:
$tmp = $i - $len;
 goto GDIStrDn  if $tmp  is  zero;
$tmp = $ptr + $i;
$ch = [$tmp].bu;
 # printable-safe run character: ASCII 32..126 excluding '"' and '\\'
$tmp = $ch - 32;
 goto GDIStrByte  if $tmp  is  neg;
$tmp = $ch - 127;
 goto GDIStrByte  if $tmp  is  nonneg;
$tmp = $ch - 34;
 goto GDIStrByte  if $tmp  is  zero;
$tmp = $ch - 92;
 goto GDIStrByte  if $tmp  is  zero;
 if $open  is  nonzero:
  $tmp = $chunk - 8;
   if $tmp  is  zero:
     call AppendByte(34);
    $open = 0;
    $chunk = 0;
   end
 end
 if $open  is  zero:
  $tmp = $kw - KW_BYTE;
   if $tmp  is  zero:
    $tmp = $cols - 3;
     if $tmp  is  nonneg:
       call AppendByte('\n');
       call AppendDataDirOp(KW_BYTE);
      $cols = 0;
     end
   end
   if $first  is  zero:
     if $cols  is  nonzero:
     call AppendByte(',');
     end
   end
  $first = 0;
   call AppendByte(34);
  $open = 1;
  $cols = $cols + 1;
 end
 call AppendByte($ch);
$chunk = $chunk + 1;
 goto GDIStrNx;
GDIStrByte:
 if $open  is  nonzero:
   call AppendByte(34);
  $open = 0;
  $chunk = 0;
 end
$tmp = $kw - KW_BYTE;
 if $tmp  is  zero:
  $tmp = $cols - 3;
```

```
    if $tmp is nonneg:
      call AppendByte('\n');
      call AppendDataDirOp(KW_BYTE);
     $cols = 0;
    end
   end
   if $first is zero:
    if $cols is nonzero:
      call AppendByte(',');
    end
   end
  $first = 0;
   call AppendHexByte($ch);
  $cols = $cols + 1;
  GDIStrNx:
  $i = $i + 1;
   goto GDIStrLp;
  GDIStrDn:
   if $open is nonzero:
    call AppendByte(34);
   end
   return;
  GDIImm:
  $val =  call NodeGet($node, ND_A);
   if $val is neg:
    call AppendHexOcta($val);
    return;
   end
   call AppendInt($val);
   return;
  GDILab:
  $off =  call NodeGet($node, ND_A);
  $len =  call NodeGet($node, ND_B);
   call GenPoolName($off, $len);
   return;
  GDINeg:
   call AppendByte('-');
  $inner =  call NodeGet($node, ND_A);
   call GenDataItem($inner, $kw);
   return;
end
```

25. Generator: procedure bodies.

GenDef emits the MMIXAL text for one procedure. It binds the procedure name to the current location, enters procedure scope, installs parameter aliases, collects local statement labels for scoped references, walks the statement list, then leaves procedure scope and emits a closing comment. The helpers in this section cover the bulk of procedure-body emission: statement dispatch, operands, assignments, stores, calls, returns, special-register moves, and immediate-loading utilities.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenDef($node):
  register $tmp, $body, $params, $outs, $slot, $p, $c;
  if $main_emitted is zero:
    if $user_loc_seen is zero:


      $tmp = &StrSegCode;
       call AppendCStr($tmp);
    end
   $main_emitted = -1;
  end
 $ProcNameOff =  call NodeGet($node, ND_A);
 $ProcNameLen =  call NodeGet($node, ND_B);
  # If this proc is literally named "Main", the procedure header itself
  # supplies |Main IS @|, so suppress the synthetic emission in |GFinal|
  # to avoid a duplicate-label assembler error.
 $tmp = $ProcNameLen - 4;
  if $tmp is zero:
   $p = $StringPool + $ProcNameOff;
   $c = [$p].bu;
   $tmp = $c - 'M';
    goto GDNotMain if $tmp is nonzero;
   $c = [$p + 1].bu;
   $tmp = $c - 'a';
    goto GDNotMain if $tmp is nonzero;
   $c = [$p + 2].bu;
   $tmp = $c - 'i';
    goto GDNotMain if $tmp is nonzero;
   $c = [$p + 3].bu;
   $tmp = $c - 'n';
    goto GDNotMain if $tmp is nonzero;
   $main_emitted = 1;
  end
GDNotMain:
 $tmp = &StrDefOpen;
  call AppendCStr($tmp);
  call GenLabelName($node);
  call AppendByte('\n');
  call GenLabelName($node);
 $tmp = &StrIsHere;
  call AppendCStr($tmp);
  call AliasEnterProc();
 $params =  call NodeGet($node, ND_C);
 $slot =  call BindParamAliases($params, 0);
 $outs =  call NodeGet($node, NODE_HEADER_SIZE);
  call CurProcOutsSet($outs);
  call CurOutStartSlotSet($slot);
 $slot =  call BindParamAliases($outs, $slot);
 $body =  call NodeGet($node, ND_D);
  call ProcLabelReset();
  call CollectProcLabels($body);
  call AnnotateProcStmtList($body);
  call GenStmtList($body);
  call AliasLeaveProc();
  call CurProcOutsSet(0);
  call CurOutStartSlotSet(0);
 $ProcNameOff = 0;
 $ProcNameLen = 0;
 $tmp = &StrDefClose;
  call AppendCStr($tmp);
  call GenLabelName($node);
```

```
    call AppendByte('\n');
    return;
  end
```

`GenStmtList` walks a statement-list head linked through `ND_NEXT` and dispatches each entry to `GenStatement`.

⟨ ***Generator procedures*** ⟩ +≡

```
  def GenStmtList($head):
   GSLLoop:
    if $head is zero:
      return;
    end
    call GenStatement($head);
   $head = call NodeGet($head, ND_NEXT);
    goto GSLLoop;
  end
```

`GenStatement` dispatches on the statement-node's `ND_KIND`. The handled cases now cover local constants, labels, register declarations, assignments and stores, calls and returns, `goto`, `trap`, `save`/`unsave`, rounding directives, wyde-immediate updates, and the structured-control forms `begin`, `if`, `while`, `loop`, and `do`. Truly unsupported kinds still fall through to a "`! stmt-kind N`" comment so the output remains self-describing.

Local `const` declarations share the exact AST form used at top level, so statement-level emission simply delegates to `GenConst`.

⟨ ***Generator procedures*** ⟩ +≡

```
  def GenStatement($node):
    register $kind, $tmp, $inner;
   $kind = call NodeGet($node, ND_KIND);
   $tmp = $kind - NK_CONST;


    goto GSCo     if $tmp is zero;
   $tmp = $kind - NK_LABELSTMT;
    goto GSLb     if $tmp is zero;
   $tmp = $kind - NK_POOL;
    goto GSPl     if $tmp is zero;
   $tmp = $kind - NK_TRAP;
    goto GSTr     if $tmp is zero;
   $tmp = $kind - NK_PRELD;
    goto GSPf     if $tmp is zero;
   $tmp = $kind - NK_SYNC;
    goto GSSy     if $tmp is zero;
   $tmp = $kind - NK_TRIP;
    goto GSTp     if $tmp is zero;
   $tmp = $kind - NK_RESUME;
    goto GSRm     if $tmp is zero;
   $tmp = $kind - NK_BEGIN;
    goto GSBg     if $tmp is zero;
   $tmp = $kind - NK_GOTO;
    goto GSGo     if $tmp is zero;
   $tmp = $kind - NK_RETURN;
    goto GSRt     if $tmp is zero;
   $tmp = $kind - NK_SAVE;
```

```
 goto GSSv     if $tmp  is  zero;
$tmp = $kind - NK_UNSAVE;
 goto GSUs     if $tmp  is  zero;
$tmp = $kind - NK_REGDECL;
 goto GSRd     if $tmp  is  zero;
$tmp = $kind - NK_ASSIGN;
 goto GSAs     if $tmp  is  zero;
$tmp = $kind - NK_STORE;
 goto GSSt     if $tmp  is  zero;
$tmp = $kind - NK_CALL;
 goto GSCl     if $tmp  is  zero;
$tmp = $kind - NK_CALLASN;
 goto GSCa     if $tmp  is  zero;
$tmp = $kind - NK_IF;
 goto GSIf     if $tmp  is  zero;
$tmp = $kind - NK_WHILE;
 goto GSWh     if $tmp  is  zero;
$tmp = $kind - NK_LOOP;
 goto GSLp     if $tmp  is  zero;
$tmp = $kind - NK_DO;
 goto GSDo     if $tmp  is  zero;
$tmp = $kind - NK_BREAK;
 goto GSBk     if $tmp  is  zero;
$tmp = $kind - NK_CONTINUE;
 goto GSCn     if $tmp  is  zero;
$tmp = $kind - NK_ROUNDING;
 goto GSRn     if $tmp  is  zero;
$tmp = $kind - NK_WYDESET;
 goto GSWs     if $tmp  is  zero;
 call GenStmtComment($kind);
 return;
GSCo:
 call GenConst($node);
 return;
GSLb:
 call GenLabelStmt($node);
 return;
GSPl:
$tmp = &StrPool;
 call AppendCStr($tmp);
 return;
GSTr:
 call GenTrap($node);
 return;
GSPf:
 call GenPreldLike($node);
 return;
GSSy:
 call GenSync($node);
 return;
GSTp:
 call GenTrip($node);
 return;
GSRm:
 call GenResume($node);
 return;
GSBg:
$inner =  call NodeGet($node, ND_A);
 call GenStmtList($inner);
 return;
GSGo:
 call GenGoto($node);
```

```
    return;
  GSRt:
    call GenReturn($node);
    return;
  GSSv:
    call GenSaveLike($node, 0);
    return;
  GSUs:
    call GenSaveLike($node, 1);
    return;
  GSRd:
    call GenRegList($node);
    return;
  GSAs:
    call GenAssign($node);
    return;
  GSSt:
    call GenStore($node);
    return;
  GSCl:
    call GenCall($node, 0);
    return;
  GSCa:
    call GenCall($node, 1);
    return;
  GSIf:
    call GenIf($node);
    return;
  GSWh:
    call GenWhile($node);
    return;
  GSLp:
    call GenLoop($node);
    return;
  GSDo:
    call GenDo($node);
    return;
  GSBk:
    call GenBreak($node);
    return;
  GSCn:
    call GenContinue($node);
    return;
  GSRn:
    call GenRounding($node);
    return;
  GSWs:
    call GenWydeSet($node);
    return;
end
```

`GenWydeSet` emits the instruction for a wyde-immediate statement. The `?=` form (`ND_D`$= 0$) lowers to `SETL`; each of the eight named wyde ops (`ND_D`$= 1$, `ND_C`$\in 1 \ldots 8$) lowers to its single-register MMIX wyde-immediate opcode: `ANDNH`, `ANDNL`, `ANDNMH`, `ANDNML`, `ORH`, `ORL`, `ORMH`, `ORML`. Destination register and 16-bit immediate are both written as the two operands of the emitted instruction.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenWydeSet($node):
  register $tmp, $dst, $rhs, $op, $flag;
 $dst  =  call NodeGet($node, ND_A);
 $rhs  =  call NodeGet($node, ND_B);


 $op   =  call NodeGet($node, ND_C);
 $flag =  call NodeGet($node, ND_D);
  if $flag  is  zero:
   $tmp = &StrOpSETL;
    call AppendCStr($tmp);
    goto GWsEmit;
  end
 $tmp = &StrIndent8;
  call AppendCStr($tmp);
 $tmp = $op - 1;
  if $tmp  is  zero: $tmp = &StrOpANDNH;   call AppendCStr($tmp);  goto GWsEmit;  end
 $tmp = $op - 2;
  if $tmp  is  zero: $tmp = &StrOpANDNL;   call AppendCStr($tmp);  goto GWsEmit;  end
 $tmp = $op - 3;
  if $tmp  is  zero: $tmp = &StrOpANDNMH;  call AppendCStr($tmp);  goto GWsEmit;  end
 $tmp = $op - 4;
  if $tmp  is  zero: $tmp = &StrOpANDNML;  call AppendCStr($tmp);  goto GWsEmit;  end
 $tmp = $op - 5;
  if $tmp  is  zero: $tmp = &StrOpORH;     call AppendCStr($tmp);  goto GWsEmit;  end
 $tmp = $op - 6;
  if $tmp  is  zero: $tmp = &StrOpORL;     call AppendCStr($tmp);  goto GWsEmit;  end
 $tmp = $op - 7;
  if $tmp  is  zero: $tmp = &StrOpORMH;    call AppendCStr($tmp);  goto GWsEmit;  end
 $tmp = &StrOpORML;
  call AppendCStr($tmp);
 GWsEmit:
  call GenRegRef($dst);
  call AppendByte(',');
  call GenOperand($rhs);
  call AppendByte('\n');
  return;
end
```

`GenRounding` emits the MMIXAL sequence that installs one of the four IEEE rounding modes into the rounding-mode subfield of the arithmetic status register `rA`. MMIX encodes the mode in bits 16–17 of `rA`: `1`=nearest, `2`=off-toward-zero, `3`=up, `4`=down. The portable pattern is "read `rA`, clear the mode bits, OR in the fresh encoding, write `rA` back," emitted as:

ĠET $255,rA
ANDNMH $255,#0003
INCMH $255,#000M
PUT rA,$255 where `M` is the mode number. A trailing comment tags each line with the mnemonic name so the output remains reader-friendly.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenRounding($node):
  register $tmp, $kw, $mode;
 $kw =  call NodeGet($node, ND_A);
 $mode = 1;
 $tmp = $kw - KW_NEAREST;  if $tmp  is  zero: $mode = 1;  end
 $tmp = $kw - KW_ZERO;     if $tmp  is  zero: $mode = 2;  end
 $tmp = $kw - KW_UP;       if $tmp  is  zero: $mode = 3;  end
 $tmp = $kw - KW_DOWN;     if $tmp  is  zero: $mode = 4;  end
 $tmp = &StrRndGet;   call AppendCStr($tmp);
 $tmp = &StrRndClr;   call AppendCStr($tmp);
 $tmp = &StrRndSetA;  call AppendCStr($tmp);
  call AppendInt($mode);
 $tmp = &StrRndSetB;  call AppendCStr($tmp);
 $tmp = &StrRndPut;   call AppendCStr($tmp);
  return;
end
```

`GenRegRef` renders a register-reference node (`NK_REG`, `NK_REGNAME`, or `NK_SREG`) as MMIXAL text. `NK_REG` writes `$N`; `NK_REGNAME` copies the pool text (which already contains the `$` sigil, possibly rewritten through the alias table); and `NK_SREG` is emitted via `GenEmitSregName` as the bare MMIX special register name (`rA`, `rJ`, ...).

⟨ ***Generator procedures*** ⟩ +≡

```
def GenRegRef($node):
  register $kind, $tmp, $v, $len;
 $kind =  call NodeGet($node, ND_KIND);
 $tmp = $kind - NK_REG;


  goto GRRNum  if $tmp  is  zero;
 $tmp = $kind - NK_REGNAME;
  goto GRRNm   if $tmp  is  zero;
 $tmp = $kind - NK_SREG;
  goto GRRSr   if $tmp  is  zero;
  call AppendByte('?');
  return;
 GRRNum:
 $v =  call NodeGet($node, ND_A);
  call AppendByte('$');
  call AppendInt($v);
  return;
 GRRNm:
 $v   =  call NodeGet($node, ND_A);
 $len =  call NodeGet($node, ND_B);
 $tmp =  call LookupAlias($v, $len);
  goto GRRNmAlias  if $tmp  is  nonzero;
  call GenRegName($v, $len);
  return;
 GRRNmAlias:
 $tmp = $tmp - 1;
  call AppendByte('$');
  call AppendInt($tmp);
  return;
 GRRSr:
 $v =  call NodeGet($node, ND_A);
  call GenEmitSregName($v);
  return;
```

```
end
```

`GenEmitSregName` walks the lexer's special-register table looking for the entry whose trailing index byte matches `$idx`, then emits the MMIXAL name (`rA`, `rBB`, ...) without a leading `$` because MMIX special registers are referenced bare in opcodes like `GET` and `PUT`. A miss falls back to `r?` so the output stays syntactically visible even when the AST carries a bogus index.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenEmitSregName($idx):
  register $p, $len, $i, $tmp, $ch;
 $p = &SregTable;
 GESLp:


 $len = [$p].bu;
  goto GESMiss  if $len  is  zero;
 $p = $p + 1;
 $tmp = $p + $len;
 $ch  = [$tmp].bu;
 $tmp = $ch - $idx;
  goto GESHit  if $tmp  is  zero;
 $p = $p + $len;
 $p = $p + 1;
  goto GESLp;
 GESHit:
  call AppendByte('r');
 $i = 0;
 GESEmit:
 $tmp = $i - $len;
  goto GESDone  if $tmp  is  zero;
 $tmp = $p + $i;
 $ch  = [$tmp].bu;
  call AppendByte($ch);
 $i = $i + 1;
  goto GESEmit;
 GESDone:
  return;
 GESMiss:
  call AppendByte('r');
  call AppendByte('?');
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def GenRegNum($reg):
  call AppendByte('$');
  call AppendInt($reg);
  return;
end
```

`GenGoto` handles both direct and indirect `goto` statements, with or without an attached condition. When a condition is present the branch is inverted and routed around the jump via a synthetic skip label, reusing the later control-flow helpers.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenGoto($node):
  register $tmp, $cond, $ind, $skip, $reg, $off;
 $cond =  call NodeGet($node, ND_C);
 $ind  =  call NodeGet($node, ND_D);


  if $cond  is  nonzero:
   $skip =  call GenNewLabel();
    call GenCondBranch($cond, $skip, 1);
  end
  if $ind  is  nonzero:
   $tmp = &StrGoInd;
    call AppendCStr($tmp);
   $reg =  call NodeGet($node, ND_A);
    call GenRegRef($reg);
   $tmp = &StrCommaZero;
    call AppendCStr($tmp);
    goto GGtEnd;
  end
 $tmp = &StrJmpDir;
  call AppendCStr($tmp);
  if $gen_in_proc  is  nonzero:
   $off =  call NodeGet($node, ND_A);
   $reg =  call NodeGet($node, ND_B);
   $tmp =  call ProcLabelMode($off, $reg);
   $tmp = $tmp - 2;
    if $tmp  is  zero:
      call GenScopedLabelName($node);
      goto GGtEnd;
    end
  end
 GGtPlain:
  call GenLabelName($node);
 GGtEnd:
  call AppendByte('\n');
  if $cond  is  nonzero:
    call GenEmitLabelDef($skip);
  end
  return;
end
```

`GenReturnBinopLoad` emits one binary-op return expression directly into the concrete register number `$dstreg`. The left operand is first loaded into `$dstreg` when it is a label or a wide immediate-like operand; the right operand is loaded into scratch register `$255` when needed.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenReturnBinopLoad($node, $dstreg):
  register $op, $flag, $base, $off, $tmp, $kind, $need_base_load, $need_off_load;
  register $off_off, $off_len;
 $op =  call NodeGet($node, ND_C);


 $flag =  call NodeGet($node, ND_D);
 $base =  call NodeGet($node, ND_A);
 $off =  call NodeGet($node, ND_B);
 $need_base_load = 0;
 $kind =  call NodeGet($base, ND_KIND);
 $tmp = $kind - NK_LABELREF;
```

```
    goto GRBLBaseWide if $tmp is zero;
   $tmp = call NeedsWideOperandLoad($base);
    goto GRBLBaseReady if $tmp is zero;
   GRBLBaseWide:
    call GenSetRegFromOperand($dstreg, $base);
   $need_base_load = 1;
   GRBLBaseReady:
   $need_off_load = 0;
   $kind = call NodeGet($off, ND_KIND);
   $tmp = $kind - NK_LABELREF;
    goto GRBLOffMaybeConst if $tmp is zero;
    goto GRBLOffWide;
   GRBLOffMaybeConst:
   $off_off = call NodeGet($off, ND_A);
   $off_len = call NodeGet($off, ND_B);
   $tmp = call GenLoadConstNameRegNum(255, $off_off, $off_len);
    goto GRBLOffLoaded if $tmp is nonzero;
   GRBLOffWide:
   $tmp = call NeedsWideOperandLoad($off);
    goto GRBLOffReady if $tmp is zero;
    call GenSetRegFromOperand(255, $off);
   GRBLOffLoaded:
   $need_off_load = 1;
   GRBLOffReady:
   $tmp = &StrIndent8;
    call AppendCStr($tmp);
    call GenBinopMnemonic($op, $flag);
   $tmp = &StrFourSpace;
    call AppendCStr($tmp);
    call GenRegNum($dstreg);
    call AppendByte(',');
    goto GRBLBaseReg if $need_base_load is nonzero;
    call GenOperand($base);
    goto GRBLBaseDone;
   GRBLBaseReg:
    call GenRegNum($dstreg);
   GRBLBaseDone:
    call AppendByte(',');
    goto GRBLOffReg if $need_off_load is nonzero;
    call GenOperand($off);
    goto GRBLOffDone;
   GRBLOffReg:
    call GenRegNum(255);
   GRBLOffDone:
    call AppendByte('\n');
    return;
  end
```

`GenReturnLoadExprLoad` emits one load-expression return directly into the concrete register number `$dstreg`. The current self-hosting uses only register/immediate address parts here, so the same direct operand emission as ordinary loads is sufficient.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenReturnLoadExprLoad($node, $dstreg):
  register $tmp, $base, $off, $sz;
 $base =  call NodeGet($node, ND_A);
 $off =  call NodeGet($node, ND_B);
 $sz =  call NodeGet($node, ND_C);
 $tmp = &StrIndent8;
  call AppendCStr($tmp);
  call GenSizeOp($sz, 0);
  call GenRegNum($dstreg);
  call AppendByte(',');
  call GenOperand($base);
  call AppendByte(',');
  if $off  is  nonzero:
    call GenOperand($off);
  end
  if $off  is  zero:
    call AppendByte('0');
  end
  call AppendByte('\n');
  return;
end
```

GenReturnExprLoad assigns one value from a `return` expression list to a declared return register identified by raw register number `$declreg`. Register-kind nodes (NK_REG, NK_REGNAME, NK_SREG) emit a `SET` instruction. NK_IMM emits an immediate load via GenLoadImmRegNum. `NK_UNOP/TK_MINUS` over an NK_IMM inner node negates the value and loads it. NK_LABELREF resolves the name as a compile-time constant alias via GenLoadConstNameRegNum. NK_BINOP emits a direct arithmetic instruction via GenReturnBinopLoad. NK_LOAD emits a direct load via GenReturnLoadExprLoad. NK_CALL emits the full call sequence via GenCall (result lands in the call-window base `$last_callbase`) then, if `$declreg` differs, emits `SET $declreg,$last_callbase`.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenReturnExprLoad($node, $declreg):
  register $kind, $tmp, $inner, $v, $off, $len;
 $kind =  call NodeGet($node, ND_KIND);
 $tmp = $kind - NK_REG;       goto GRELReg  if $tmp  is  zero;


 $tmp = $kind - NK_REGNAME;   goto GRELReg  if $tmp  is  zero;
 $tmp = $kind - NK_SREG;      goto GRELReg  if $tmp  is  zero;
 $tmp = $kind - NK_IMM;       goto GRELImm  if $tmp  is  zero;
 $tmp = $kind - NK_UNOP;      goto GRELNeg  if $tmp  is  zero;
 $tmp = $kind - NK_LABELREF;  goto GRELLab   if $tmp  is  zero;
 $tmp = $kind - NK_BINOP;     goto GRELBin   if $tmp  is  zero;
 $tmp = $kind - NK_LOAD;      goto GRELLoad  if $tmp  is  zero;
 $tmp = $kind - NK_CALL;      goto GRELCall  if $tmp  is  zero;
  call AppendByte('?');
  return;
 GRELReg:
 $tmp = &StrOpSET;
  call AppendCStr($tmp);
  call GenRegNum($declreg);
  call AppendByte(',');
  call GenRegRef($node);
  call AppendByte('\n');
  return;
```

```
  GRELImm:
  $v =  call NodeGet($node, ND_A);
   call GenLoadImmRegNum($declreg, $v);
   return;
  GRELNeg:
  $inner =  call NodeGet($node, ND_A);
  $v     =  call NodeGet($inner, ND_A);
  $v     = -$v;
   call GenLoadImmRegNum($declreg, $v);
   return;
  GRELLab:
  $off =  call NodeGet($node, ND_A);
  $len =  call NodeGet($node, ND_B);
   call GenLoadConstNameRegNum($declreg, $off, $len);
   return;
  GRELBin:
   call GenReturnBinopLoad($node, $declreg);
   return;
  GRELLoad:
   call GenReturnLoadExprLoad($node, $declreg);
   return;
  GRELCall:
   call GenCall($node, 0);
   # Result lands in $last_callbase; move to declared return reg if different.
  $tmp = $declreg - $last_callbase;
   goto GRELCallDone  if $tmp  is  zero;
  $tmp = &StrOpSET;
   call AppendCStr($tmp);
   call GenRegNum($declreg);
   call AppendByte(',');
   call GenRegNum($last_callbase);
   call AppendByte('\n');
  GRELCallDone:
   return;
 end
```

GenReturn renders every shape of `return;`: `return;` (bare), `return $a, $b;` (with results), and each of those two with a trailing `if cond` guard. The core instruction is `POP n,0` where `n` is the number of result registers. When a condition is present, the `POP` is guarded by an inverted conditional branch that hops over it to a synthetic skip label.

When all items in the return list are register-kind nodes (NK_REG/NK_REGNAME/NK_SREG), the existing shuffling path is used: values are saved through temporaries `$20..$20+n-1` then moved into `$0..$n-1` before `POP`.

When any item is a value-expression node (for example NK_IMM, NK_UNOP, NK_LABELREF, or NK_BINOP), the value-expression path runs instead. It walks the return list in parallel with `$CurProcOuts` (the current proc's declared out-parameter list) to assign each expression to its corresponding declared return register, then performs the same temp-shuffle into `$0..$n-1` using the declared register numbers as sources.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenReturn($node):
  register $tmp, $regs, $cond, $n, $cur, $first, $lSkip, $idx, $tempreg, $tempbase;
  register $outp, $oslot, $flag, $declreg, $nout;
 $regs = call NodeGet($node, ND_A);


 $cond = call NodeGet($node, ND_C);
 $lSkip = 0;
  if $cond is nonzero:
   $lSkip = call GenNewLabel();
    call GenCondBranch($cond, $lSkip, 1);
  end
  # Count $n
 $n = 0;
 $cur = $regs;
GRtCnt:
  goto GRtCntDn if $cur is zero;
 $n = $n + 1;
 $cur = call NodeGet($cur, ND_NEXT);
  goto GRtCnt;
GRtCntDn:
 $outp = call CurProcOutsGet();
  if $outp is zero:
    goto GRtDetectStart;
  end
 $nout = 0;
GRtOutCnt:
  goto GRtOutCntDn if $outp is zero;
 $nout = $nout + 1;
 $outp = call NodeGet($outp, ND_NEXT);
  goto GRtOutCnt;
GRtOutCntDn:
GRtDetectStart:
  # Save/shuffle temporaries must live above the live local frame.
 $tmp = $GlobalRegUsed;
  if $gen_in_proc is nonzero:
   $tmp = $LocalRegUsed;
  end
 $tempbase = call MaxRegUsed($tmp);
 $tempbase = $tempbase + 1;
  # Detect value-expression mode: any non-register-kind item triggers it
 $cur = $regs;
GRtDetect:
  goto GRtRegMode if $cur is zero;
 $tmp = call NodeGet($cur, ND_KIND);
 $flag = $tmp - NK_REG;      goto GRtDetNext if $flag is zero;
 $flag = $tmp - NK_REGNAME;  goto GRtDetNext if $flag is zero;
 $flag = $tmp - NK_SREG;     goto GRtDetNext if $flag is zero;
  goto GRtVEMode;    # non-register kind found
GRtDetNext:
 $cur = call NodeGet($cur, ND_NEXT);
  goto GRtDetect;
GRtRegMode:
  # -- Register mode: existing behaviour --
  if $regs is nonzero:
   $tmp = &StrCallRet;
    call AppendCStr($tmp);
   $cur  = $regs;
   $first = 1;
   GRtList:
    goto GRtListDn if $cur is zero;
```

```
    if $first is zero:
      call AppendByte(',');
      call AppendByte(' ');
    end
    call GenRegRef($cur);
   $first = 0;
   $cur = call NodeGet($cur, ND_NEXT);
    goto GRtList;
   GRtListDn:
    call AppendByte('\n');
   $cur = $regs;
   $idx = 0;
   GRtSave:
    goto GRtLoad if $cur is zero;
   $tmp = &StrOpSET;
    call AppendCStr($tmp);
   $tempreg = $tempbase;
   $tempreg = $tempreg + $idx;
    call GenRegNum($tempreg);
    call AppendByte(',');
    call GenRegRef($cur);
    call AppendByte('\n');
   $idx = $idx + 1;
   $cur = call NodeGet($cur, ND_NEXT);
    goto GRtSave;
   GRtLoad:
   $idx = 0;
   GRtLoadLoop:
   $tmp = $idx - $n;
    goto GRtLoadDone if $tmp is nonneg;
   $tmp = &StrOpSET;
    call AppendCStr($tmp);
    call GenRegNum($idx);
    call AppendByte(',');
   $tempreg = $tempbase;
   $tempreg = $tempreg + $idx;
    call GenRegNum($tempreg);
    call AppendByte('\n');
   $idx = $idx + 1;
    goto GRtLoadLoop;
   GRtLoadDone:
  end
  goto GRtPOP;
 GRtVEMode:
  # -- Value-expression mode: assign values to declared return regs, then shuffle --
  if $regs is nonzero:
    # Phase 1: assign each value-expression to its declared return register
   $outp = call CurProcOutsGet();
   $oslot = call CurOutStartSlotGet();
   $cur = $regs;
   GRtVEAssign:
    goto GRtVESavePhase if $cur is zero;
   $flag = call NodeGet($outp, ND_D);
    if $flag is zero:
     $declreg = $oslot;
   else:
     $declreg = call NodeGet($outp, ND_C);
    end
   $oslot = $oslot + 1;
    call GenReturnExprLoad($cur, $declreg);
   $outp = call NodeGet($outp, ND_NEXT);
   $cur = call NodeGet($cur, ND_NEXT);
```

```
      goto GRtVEAssign;
    GRtVESavePhase:
      # Phase 2: save declared return regs to temporaries
    $outp  =  call CurProcOutsGet();
    $oslot =  call CurOutStartSlotGet();
    $idx   = 0;
    GRtVESave:
      goto GRtVELoad  if $outp  is  zero;
    $flag =  call NodeGet($outp, ND_D);
      if $flag  is  zero:
        $declreg = $oslot;
    else:
        $declreg =  call NodeGet($outp, ND_C);
      end
    $oslot = $oslot + 1;
    $tmp = &StrOpSET;
      call AppendCStr($tmp);
    $tempreg = $tempbase;
    $tempreg = $tempreg + $idx;
      call GenRegNum($tempreg);
      call AppendByte(',');
      call GenRegNum($declreg);
      call AppendByte('\n');
    $idx  = $idx + 1;
    $outp =  call NodeGet($outp, ND_NEXT);
      goto GRtVESave;
    GRtVELoad:
      # Phase 3: move temporaries into $0..$n-1 for POP
    $idx = 0;
    GRtVELoadLoop:
    $tmp = $idx - $n;
      goto GRtVELoadDone  if $tmp  is  nonneg;
    $tmp = &StrOpSET;
      call AppendCStr($tmp);
      call GenRegNum($idx);
      call AppendByte(',');
    $tempreg = $tempbase;
    $tempreg = $tempreg + $idx;
      call GenRegNum($tempreg);
      call AppendByte('\n');
    $idx = $idx + 1;
      goto GRtVELoadLoop;
    GRtVELoadDone:
   end
  GRtPOP:
  $tmp = &StrIndent8;
   call AppendCStr($tmp);
  $tmp = &StrOpPOP;
   call AppendCStr($tmp);
   call AppendInt($n);
  $tmp = &StrCommaZero;
   call AppendCStr($tmp);
   if $lSkip  is  nonzero:
     call GenEmitLabelDef($lSkip);
   end
   return;
 end
```

`GenSaveLike` shares code between `NK_SAVE` (`$unsave`= 0) and `NK_UNSAVE` (`$unsave`= 1). The target register is in `ND_A`. MMIXAL syntax is `SAVE $r,0` and `UNSAVE 0,$r`.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenSaveLike($node, $unsave):
  register $tmp, $reg;
 $reg =  call NodeGet($node, ND_A);
  if $unsave  is  nonzero:
   $tmp = &StrUnsaveDir;
    call AppendCStr($tmp);
    call GenRegRef($reg);
    call AppendByte('\n');
    return;
  end
 $tmp = &StrSaveDir;
  call AppendCStr($tmp);
  call GenRegRef($reg);
 $tmp = &StrCommaZero;
  call AppendCStr($tmp);
  return;
end
```

GenRegList emits a `register` declaration as a binding pass for named registers. A single-name top-level alias that targets an explicit register is bound directly; otherwise the routine allocates a free local or global register, records the alias, and emits the top-level alias definition when needed.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenRegList($node):
  register $head, $next, $tmp, $local, $reg, $off, $len, $used, $target;
 $head =  call NodeGet($node, ND_A);
 $local = $gen_in_proc;


  if $head  is  zero:
    return;
  end
 $next =  call NodeGet($head, ND_NEXT);
  if $next  is  zero:
    goto GRLLoop;
  end
 $tmp =  call NodeGet($head, ND_KIND);
 $tmp = $tmp - NK_REGNAME;
  goto GRLLoop  if $tmp  is  nonzero;
 $tmp =  call NodeGet($next, ND_NEXT);
  goto GRLLoop  if $tmp  is  nonzero;
 $target =  call ResolveAliasTarget($next);
  goto GRLLoop  if $target  is  zero;
 $reg = $target - 1;
 $off =  call NodeGet($head, ND_A);
 $len =  call NodeGet($head, ND_B);
  call BindAlias($local, $off, $len, $reg);
  if $local  is  zero:
    call EmitAliasDef($off, $len, $reg);
  end
  return;
GRLLoop:
  if $head  is  zero:
    return;
  end
 $used = $GlobalRegUsed;
```

```
    if $local is nonzero:
     $used = $LocalRegUsed;
    end
   $tmp = call NodeGet($head, ND_KIND);
   $tmp = $tmp - NK_REGNAME;
    goto GRLName if $tmp is zero;
   $tmp = call NodeGet($head, ND_KIND);
   $tmp = $tmp - NK_REG;
    goto GRLNum if $tmp is zero;
    goto GRLNext;
   GRLName:
   $off = call NodeGet($head, ND_A);
   $len = call NodeGet($head, ND_B);
   $tmp = call AllocFreeReg($used, 0);
   $reg = $tmp - 1;
    call BindAlias($local, $off, $len, $reg);
    if $local is zero:
      call EmitAliasDef($off, $len, $reg);
    end
    goto GRLNext;
   GRLNum:
   $reg = call NodeGet($head, ND_A);
    call MarkRegUsed($used, $reg);
   GRLNext:
   $head = call NodeGet($head, ND_NEXT);
    goto GRLLoop;
  end
```

`GenOperand` renders an "atom" expression node—one that can legally appear as a source operand in a single MMIX instruction. The accepted kinds are registers (`NK_REG`, `NK_REGNAME`, `NK_SREG`), integer immediates (`NK_IMM`), bare label references (`NK_LABELREF`), and address-of (`NK_GETA`). `NK_UNOP` is accepted only in the `TK_MINUS`-applied-to-`NK_IMM` shape, producing a negated decimal literal. Everything else emits `?` so malformed input remains visible without derailing the surrounding instruction.

⟨ ***Generator procedures*** ⟩ +≡

```
  def GenOperand($node):
    register $kind, $tmp, $v, $len, $op, $inner, $innerkind;
   $kind = call NodeGet($node, ND_KIND);
   $tmp = $kind - NK_REG;


    goto GOpRg if $tmp is zero;
   $tmp = $kind - NK_REGNAME;
    goto GOpRg if $tmp is zero;
   $tmp = $kind - NK_SREG;
    goto GOpRg if $tmp is zero;
   $tmp = $kind - NK_IMM;
    goto GOpIm if $tmp is zero;
   $tmp = $kind - NK_FLOAT;
    goto GOpFl if $tmp is zero;
   $tmp = $kind - NK_LABELREF;
    goto GOpNm if $tmp is zero;
   $tmp = $kind - NK_GETA;
    goto GOpGa if $tmp is zero;
   $tmp = $kind - NK_UNOP;
    goto GOpUn if $tmp is zero;
    call AppendByte('?');
```

```
 return;
GOpRg:
 call GenRegRef($node);
 return;
GOpIm:
$v =  call NodeGet($node, ND_A);
 if $v  is  neg:
   call AppendHexOcta($v);
   return;
 end
 call AppendInt($v);
 return;
GOpFl:
$v   =  call NodeGet($node, ND_A);
$len =  call NodeGet($node, ND_B);
$tmp = $StringPool + $v;
 call AppendBytes($tmp, $len);
 return;
GOpNm:
$v   =  call NodeGet($node, ND_A);
$len =  call NodeGet($node, ND_B);
 call GenPoolName($v, $len);
 return;
GOpGa:
$v   =  call NodeGet($node, ND_A);
$len =  call NodeGet($node, ND_B);
 if $gen_in_proc  is  nonzero:
  $tmp =  call NodeGet($node, ND_C);
   if $tmp  is  nonzero:
    $tmp = $tmp - 2;
     if $tmp  is  zero:
       call GenScopedLabelPoolName($v, $len);
       return;
     end
     call GenPoolName($v, $len);
     return;
   end
 end
$tmp =  call IsDeclaredDataLabelName($v, $len);
 if $tmp  is  nonzero:
  $tmp =  call IsDataLabelAddrTaken($v, $len);
   if $tmp  is  nonzero:
     call EmitDataGregName($v, $len);
   end
   if $tmp  is  zero:
     call GenPoolName($v, $len);
   end
   return;
 end
$tmp =  call IsDataLabel($v, $len);
 if $tmp  is  nonzero:
  $tmp =  call IsDataLabelAddrTaken($v, $len);
   if $tmp  is  nonzero:
     call EmitDataGregName($v, $len);
   end
   if $tmp  is  zero:
     call GenPoolName($v, $len);
   end
   return;
 end
 call GenPoolName($v, $len);
 return;
```

```
GOpUn:
$op = call NodeGet($node, ND_C);
$tmp = $op - TK_MINUS;
 if $tmp is zero:
  $inner = call NodeGet($node, ND_A);
  $innerkind = call NodeGet($inner, ND_KIND);
  $tmp = $innerkind - NK_IMM;
   if $tmp is zero:
     call AppendByte('-');
    $v = call NodeGet($inner, ND_A);
     call AppendInt($v);
     return;
   end
 end
$tmp = $op - TK_MINUSU;
 if $tmp is zero:
   call AppendByte('-');
   call AppendByte('u');
   call AppendByte(' ');
  $inner = call NodeGet($node, ND_A);
   call GenOperand($inner);
   return;
 end
 call AppendByte('?');
 return;
end
```

`GenBinopMnemonic` writes the MMIXAL arithmetic opcode for an `NK_BINOP` whose op code sits in `ND_C` and whose `ND_D` flag distinguishes token-encoded (0) from keyword-encoded (1) operators. Only the four core arithmetic tokens are rendered concretely for now; keyword-encoded binary ops and the comparison family emit a `???` placeholder that keeps the line parseable.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenBinopMnemonic($op, $flag):
  register $tmp;
  if $flag is nonzero:
   $tmp = $op - KW_ANDN;


    if $tmp is zero: $tmp = &StrOpANDN;   call AppendCStr($tmp); return; end
   $tmp = $op - KW_ORN;
    if $tmp is zero: $tmp = &StrOpORN;    call AppendCStr($tmp); return; end
   $tmp = $op - KW_NAND;
    if $tmp is zero: $tmp = &StrOpNAND;   call AppendCStr($tmp); return; end
   $tmp = $op - KW_NOR;
    if $tmp is zero: $tmp = &StrOpNOR;    call AppendCStr($tmp); return; end
   $tmp = $op - KW_NXOR;
    if $tmp is zero: $tmp = &StrOpNXOR;   call AppendCStr($tmp); return; end
   $tmp = $op - KW_MOR;
    if $tmp is zero: $tmp = &StrOpMOR;    call AppendCStr($tmp); return; end
   $tmp = $op - KW_MXOR;
    if $tmp is zero: $tmp = &StrOpMXOR;   call AppendCStr($tmp); return; end
   $tmp = $op - KW_MUX;
    if $tmp is zero: $tmp = &StrOpMUX;    call AppendCStr($tmp); return; end
   $tmp = $op - KW_SADD;
    if $tmp is zero: $tmp = &StrOpSADD;   call AppendCStr($tmp); return; end
   $tmp = $op - KW_BDIF;
    if $tmp is zero: $tmp = &StrOpBDIF;   call AppendCStr($tmp); return; end
   $tmp = $op - KW_WDIF;
```

```
    if $tmp is zero: $tmp = &StrOpWDIF;   call AppendCStr($tmp); return; end
   $tmp = $op - KW_TDIF;
    if $tmp is zero: $tmp = &StrOpTDIF;   call AppendCStr($tmp); return; end
   $tmp = $op - KW_ODIF;
    if $tmp is zero: $tmp = &StrOpODIF;   call AppendCStr($tmp); return; end
   $tmp = $op - KW_CMP;
    if $tmp is zero: $tmp = &StrOpCMP;    call AppendCStr($tmp); return; end
   $tmp = $op - KW_CMPU;
    if $tmp is zero: $tmp = &StrOpCMPU;   call AppendCStr($tmp); return; end
   $tmp = $op - KW_CMPEDOT;
    if $tmp is zero: $tmp = &StrOpFCMPE;  call AppendCStr($tmp); return; end
   $tmp = $op - KW_EQLEDOT;
    if $tmp is zero: $tmp = &StrOpFEQLE;  call AppendCStr($tmp); return; end
   $tmp = $op - KW_UNEDOT;
    if $tmp is zero: $tmp = &StrOpFUNE;   call AppendCStr($tmp); return; end
    goto GBMStub;
 end
$tmp = $op - TK_PLUS;
 if $tmp is zero:
  $tmp = &StrOpADD;
   call AppendCStr($tmp);
   return;
 end
$tmp = $op - TK_PLUSU;
 if $tmp is zero:
  $tmp = &StrOpADDU;
   call AppendCStr($tmp);
   return;
 end
$tmp = $op - TK_MINUS;
 if $tmp is zero:
  $tmp = &StrOpSUB;
   call AppendCStr($tmp);
   return;
 end
$tmp = $op - TK_MINUSU;
 if $tmp is zero:
  $tmp = &StrOpSUBU;
   call AppendCStr($tmp);
   return;
 end
$tmp = $op - TK_STAR;
 if $tmp is zero:
  $tmp = &StrOpMUL;
   call AppendCStr($tmp);
   return;
 end
$tmp = $op - TK_STARU;
 if $tmp is zero:
  $tmp = &StrOpMULU;
   call AppendCStr($tmp);
   return;
 end
$tmp = $op - TK_SLASH;
 if $tmp is zero:
  $tmp = &StrOpDIV;
   call AppendCStr($tmp);
   return;
 end
$tmp = $op - TK_SLASHU;
 if $tmp is zero:
  $tmp = &StrOpDIVU;
```

```
    call AppendCStr($tmp);
    return;
  end
 $tmp = $op - TK_FADD;
  if $tmp  is  zero: $tmp = &StrOpFADD;  call AppendCStr($tmp);  return;  end
 $tmp = $op - TK_FSUB;
  if $tmp  is  zero: $tmp = &StrOpFSUB;  call AppendCStr($tmp);  return;  end
 $tmp = $op - TK_FMUL;
  if $tmp  is  zero: $tmp = &StrOpFMUL;  call AppendCStr($tmp);  return;  end
 $tmp = $op - TK_FDIV;
  if $tmp  is  zero: $tmp = &StrOpFDIV;  call AppendCStr($tmp);  return;  end
 $tmp = $op - TK_FREM;
  if $tmp  is  zero: $tmp = &StrOpFREM;  call AppendCStr($tmp);  return;  end
 $tmp = $op - TK_AMP;
  if $tmp  is  zero: $tmp = &StrOpAND;   call AppendCStr($tmp);  return;  end
 $tmp = $op - TK_PIPE;
  if $tmp  is  zero: $tmp = &StrOpOR;    call AppendCStr($tmp);  return;  end
 $tmp = $op - TK_CARET;
  if $tmp  is  zero: $tmp = &StrOpXOR;   call AppendCStr($tmp);  return;  end
 $tmp = $op - TK_SHL;
  if $tmp  is  zero: $tmp = &StrOpSL;    call AppendCStr($tmp);  return;  end
 $tmp = $op - TK_SHR;
  if $tmp  is  zero: $tmp = &StrOpSR;    call AppendCStr($tmp);  return;  end
 $tmp = $op - TK_SHLU;
  if $tmp  is  zero: $tmp = &StrOpSLU;   call AppendCStr($tmp);  return;  end
 $tmp = $op - TK_SHRU;
  if $tmp  is  zero: $tmp = &StrOpSRU;   call AppendCStr($tmp);  return;  end
 $tmp = $op - TK_2ADDU;
  if $tmp  is  zero: $tmp = &StrOp2ADDU;   call AppendCStr($tmp);  return;  end
 $tmp = $op - TK_4ADDU;
  if $tmp  is  zero: $tmp = &StrOp4ADDU;   call AppendCStr($tmp);  return;  end
 $tmp = $op - TK_8ADDU;
  if $tmp  is  zero: $tmp = &StrOp8ADDU;   call AppendCStr($tmp);  return;  end
 $tmp = $op - TK_16ADDU;
  if $tmp  is  zero: $tmp = &StrOp16ADDU;  call AppendCStr($tmp);  return;  end
 GBMStub:
  call AppendByte('?');
  call AppendByte('?');
  call AppendByte('?');
  return;
end
```

`GenSizeOp` writes the load (`$isStore`= 0) or store (`$isStore`= 1) mnemonic for the size code stored in `NK_LOAD.ND_C` or `NK_STORE.ND_C`. The codes are `1`= b, `2`=bu, `3`= w, `4`=wu, `5`= t, `6`=tu, `7`= o, `8`=sf, `9`= ht, `10`=unc. Each branch writes the full 8-wide opcode column (mnemonic + padding) so the caller can follow directly with the first operand.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenSizeOp($size, $isStore):
  register $tmp;
  if $isStore is nonzero:
    goto GSOStore;


  end
 $tmp = $size - 1;
  if $tmp is zero: $tmp = &StrLDB;     call AppendCStr($tmp); return; end
 $tmp = $size - 2;
  if $tmp is zero: $tmp = &StrLDBU;    call AppendCStr($tmp); return; end
 $tmp = $size - 3;
  if $tmp is zero: $tmp = &StrLDW;     call AppendCStr($tmp); return; end
 $tmp = $size - 4;
  if $tmp is zero: $tmp = &StrLDWU;    call AppendCStr($tmp); return; end
 $tmp = $size - 5;
  if $tmp is zero: $tmp = &StrLDT;     call AppendCStr($tmp); return; end
 $tmp = $size - 6;
  if $tmp is zero: $tmp = &StrLDTU;    call AppendCStr($tmp); return; end
 $tmp = $size - 7;
  if $tmp is zero: $tmp = &StrLDO;     call AppendCStr($tmp); return; end
 $tmp = $size - 8;
  if $tmp is zero: $tmp = &StrLDSF;    call AppendCStr($tmp); return; end
 $tmp = $size - 9;
  if $tmp is zero: $tmp = &StrLDHT;    call AppendCStr($tmp); return; end
 $tmp = &StrLDUNC;
  call AppendCStr($tmp);
  return;
 GSOStore:
 $tmp = $size - 1;
  if $tmp is zero: $tmp = &StrSTB;     call AppendCStr($tmp); return; end
 $tmp = $size - 2;
  if $tmp is zero: $tmp = &StrSTBU;    call AppendCStr($tmp); return; end
 $tmp = $size - 3;
  if $tmp is zero: $tmp = &StrSTW;     call AppendCStr($tmp); return; end
 $tmp = $size - 4;
  if $tmp is zero: $tmp = &StrSTWU;    call AppendCStr($tmp); return; end
 $tmp = $size - 5;
  if $tmp is zero: $tmp = &StrSTT;     call AppendCStr($tmp); return; end
 $tmp = $size - 6;
  if $tmp is zero: $tmp = &StrSTTU;    call AppendCStr($tmp); return; end
 $tmp = $size - 7;
  if $tmp is zero: $tmp = &StrSTO;     call AppendCStr($tmp); return; end
 $tmp = $size - 8;
  if $tmp is zero: $tmp = &StrSTSF;    call AppendCStr($tmp); return; end
 $tmp = $size - 9;
  if $tmp is zero: $tmp = &StrSTHT;    call AppendCStr($tmp); return; end
 $tmp = &StrSTUNC;
  call AppendCStr($tmp);
  return;
end
```

Per-size load/store mnemonic strings, each padded to the standard 8-character opcode column.

⟨ ***Data segment*** ⟩ +≡

```
byte StrLDB   "LDB     ", 0;
byte StrLDBU  "LDBU    ", 0;
byte StrLDW   "LDW     ", 0;
byte StrLDWU  "LDWU    ", 0;
byte StrLDT   "LDT     ", 0;
byte StrLDTU  "LDTU    ", 0;
byte StrLDO   "LDO     ", 0;
byte StrLDSF  "LDSF    ", 0;
byte StrLDHT  "LDHT    ", 0;
byte StrLDUNC "LDUNC   ", 0;
byte StrSTB   "STB     ", 0;
byte StrSTBU  "STBU    ", 0;
byte StrSTW   "STW     ", 0;
byte StrSTWU  "STWU    ", 0;
byte StrSTT   "STT     ", 0;
byte StrSTTU  "STTU    ", 0;
byte StrSTO   "STO     ", 0;
byte StrSTSF  "STSF    ", 0;
byte StrSTHT  "STHT    ", 0;
byte StrSTUNC "STUNC   ", 0;
```

`GenAssign` emits the instruction implementing `$dst = RHS;`. Supported RHS shapes, each mapped to a single MMIX instruction:

- • register copy (`SET`),
- • integer immediate (using the immediate-loading helpers when a single `SETL` is not enough),
- • `&Label` or bare label (`GETA`),
- • load expression (`LDx $dst,base,off`),
- • binary-op expression (`ADD`/`SUB`/`MUL`/`DIV`).

Every other RHS shape emits a `! assign (kind N) not-yet-implemented` stub so the output line stays self-describing.

For binary operators the generator aggressively reuses registers: if a const-name base must first be materialized, it is loaded into `$dst` and that same physical register becomes the live base operand for the final instruction. Any later wide-offset materialization must respect that ownership and avoid clobbering the base before the opcode is emitted.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenAssign($node):
  register $tmp, $dst, $rhs, $rhskind, $op, $flag, $base, $off, $sz, $v, $len, $basekind, $offkind,
    $offreg, $basereg, $dstreg, $scratchreg, $probe;
 $dst = call NodeGet($node, ND_A);
 $rhs = call NodeGet($node, ND_B);


 $rhskind = call NodeGet($rhs, ND_KIND);
 $tmp = $rhskind - NK_BINOP;
  goto GAsBin if $tmp is zero;
 $tmp = $rhskind - NK_LOAD;
  goto GAsLd if $tmp is zero;
 $tmp = $rhskind - NK_GETA;
  goto GAsGa if $tmp is zero;
 $tmp = $rhskind - NK_LABELREF;
  goto GAsGa if $tmp is zero;
 $tmp = $rhskind - NK_IMM;
```

```
 goto GAsIm   if $tmp  is  zero;
$tmp = $rhskind - NK_REG;
 goto GAsSet  if $tmp  is  zero;
$tmp = $rhskind - NK_REGNAME;
 goto GAsSet  if $tmp  is  zero;
$tmp = $rhskind - NK_SREG;
 goto GAsSet  if $tmp  is  zero;
$tmp = $rhskind - NK_UNOP;
 goto GAsUn   if $tmp  is  zero;
$tmp = &StrAsnStub;
 call AppendCStr($tmp);
 call AppendInt($rhskind);
 call AppendByte('\n');
 return;
GAsSet:
 call GenSetTargetFromOperand($dst, $rhs);
 return;
GAsIm:
$v =  call NodeGet($rhs, ND_A);
 call GenLoadImm($dst, $v);
 return;
GAsGa:
$v =  call NodeGet($rhs, ND_A);
$len =  call NodeGet($rhs, ND_B);
 if $gen_in_proc  is  nonzero:
  $tmp =  call NodeGet($rhs, ND_C);
   if $tmp  is  nonzero:
    $tmp = &StrOpGETA;
     goto GAsGaOp;
   end
 end
$tmp =  call IsDeclaredDataLabelName($v, $len);
 if $tmp  is  nonzero:
  $tmp =  call IsDataLabelAddrTaken($v, $len);
   if $tmp  is  zero:
     call EmitDataGregBinding($v, $len);
   end
  $tmp = &StrOpSET;
   call AppendCStr($tmp);
   call GenRegRef($dst);
   call AppendByte(',');
   call EmitDataGregName($v, $len);
   call AppendByte('\n');
   return;
 end
$tmp =  call GenLoadConstName($dst, $v, $len);
 if $tmp  is  nonzero:
   return;
 end
$tmp =  call IsDataLabel($v, $len);
 if $tmp  is  nonzero:
  $tmp =  call LookupConstExpr($v, $len);
   if $tmp  is  zero:
     goto GAsGaDataAlias;
   end
  $tmp =  call IsConstExprNode($tmp);
   if $tmp  is  nonzero:
     goto GAsGaDataMode;
   end
  GAsGaDataAlias:
  $tmp =  call IsDataLabelAddrTaken($v, $len);
   if $tmp  is  zero:
```

```
      call EmitDataGregBinding($v, $len);
    end
   $tmp = &StrOpSET;
    call AppendCStr($tmp);
    call GenRegRef($dst);
    call AppendByte(',');
    call EmitDataGregName($v, $len);
    call AppendByte('\n');
    return;
   GAsGaDataMode:
   $tmp =  call IsDataLabelAddrTaken($v, $len);
    if $tmp  is  nonzero:
     $tmp = &StrOpSET;
    end
    if $tmp  is  zero:
     $tmp = &StrOpGETA;
    end
    goto GAsGaOp;
  end
 $tmp =  call IsConstName($v, $len);
  if $tmp  is  nonzero:
   $tmp = &StrOpSET;
    goto GAsGaOp;
  end
 $tmp =  call IsGregName($v, $len);
  if $tmp  is  nonzero:
   $tmp = &StrOpSET;
    goto GAsGaOp;
  end
 $tmp = &StrOpGETA;
 GAsGaOp:
  call AppendCStr($tmp);
  call GenRegRef($dst);
  call AppendByte(',');
  call GenOperand($rhs);
  call AppendByte('\n');
  return;
 GAsLd:
 $base =  call NodeGet($rhs, ND_A);
 $off  =  call NodeGet($rhs, ND_B);
 $sz   =  call NodeGet($rhs, ND_C);
 $tmp = &StrIndent8;
  call AppendCStr($tmp);
  call GenSizeOp($sz, 0);
  call GenRegRef($dst);
  call AppendByte(',');
  call GenOperand($base);
  if $off  is  nonzero:
    call AppendByte(',');
    call GenOperand($off);
  end
  call AppendByte('\n');
  return;
 GAsUn:
  call GenUnopAssign($dst, $rhs);
  return;
 GAsBin:
 $op   =  call NodeGet($rhs, ND_C);
 $flag =  call NodeGet($rhs, ND_D);
 $base =  call NodeGet($rhs, ND_A);
 $off  =  call NodeGet($rhs, ND_B);
 $offreg = 0;
```

```
$basereg = -1;
$dstreg = -1;
$scratchreg = 255;
$basekind =  call NodeGet($base, ND_KIND);
$tmp = $basekind - NK_REG;
 goto GAsBinBaseReg  if $tmp  is  zero;
$tmp = $basekind - NK_REGNAME;
 goto GAsBinBaseName  if $tmp  is  zero;
 goto GAsBinBaseResolved;
GAsBinBaseReg:
$basereg =  call NodeGet($base, ND_A);
 goto GAsBinBaseResolved;
GAsBinBaseName:
$v =  call NodeGet($base, ND_A);
$len =  call NodeGet($base, ND_B);
$basereg =  call LookupAlias($v, $len);
 if $basereg  is  nonzero:
  $basereg = $basereg - 1;
 end
GAsBinBaseResolved:
$tmp =  call NodeGet($dst, ND_KIND);
$v = $tmp - NK_REG;
 goto GAsBinDstReg  if $v  is  zero;
$v = $tmp - NK_REGNAME;
 goto GAsBinDstName  if $v  is  zero;
 goto GAsBinDstResolved;
GAsBinDstReg:
$dstreg =  call NodeGet($dst, ND_A);
 goto GAsBinDstResolved;
GAsBinDstName:
$v =  call NodeGet($dst, ND_A);
$len =  call NodeGet($dst, ND_B);
$dstreg =  call LookupAlias($v, $len);
 if $dstreg  is  nonzero:
  $dstreg = $dstreg - 1;
 end
GAsBinDstResolved:
$tmp = $basekind - NK_LABELREF;
 goto GAsBinBaseDone  if $tmp  is  nonzero;
$v =  call NodeGet($base, ND_A);
$len =  call NodeGet($base, ND_B);
$tmp =  call IsConstName($v, $len);
 if $tmp  is  zero:
   goto GAsBinBaseDone;
 end
$tmp =  call GenLoadConstName($dst, $v, $len);
 if $tmp  is  zero:
   call GenSetTargetFromOperand($dst, $base);
 end
$base = $dst;
GAsBinBaseDone:
$offkind =  call NodeGet($off, ND_KIND);
$tmp = $offkind - NK_LABELREF;
 goto GAsBinOffDone  if $tmp  is  nonzero;
$v =  call NodeGet($off, ND_A);
$len =  call NodeGet($off, ND_B);
$tmp =  call IsConstName($v, $len);
 if $tmp  is  zero:
   goto GAsBinOffDone;
 end
$tmp = $basereg - 255;
 if $tmp  is  zero:
```

```
    $tmp = call ResolveAliasTarget($dst);
    $probe = $tmp - 255;
     if $probe is zero:
       goto GAsBinOffScratch;
     end
    $probe = $tmp - 256;
     if $probe is zero:
       goto GAsBinOffScratch;
     end
    $tmp = call GenLoadConstName($dst, $v, $len);
     if $tmp is zero:
       call GenSetTargetFromOperand($dst, $off);
     end
    $offreg = 2;
     goto GAsBinOffDone;
   end
GAsBinOffScratch:
$tmp = call GenLoadConstNameRegNum($scratchreg, $v, $len);
 if $tmp is zero:
   call GenSetRegFromOperand($scratchreg, $off);
 end
$offreg = 1;
 goto GAsBinOffDone;
GAsBinOffWide:
$tmp = $basereg - 255;
 if $tmp is zero:
  $tmp = call ResolveAliasTarget($dst);
  $probe = $tmp - 255;
   if $probe is zero:
     goto GAsBinOffWideScratch;
   end
  $probe = $tmp - 256;
   if $probe is zero:
     goto GAsBinOffWideScratch;
   end
   call GenSetRegFromOperand($dstreg, $off);
  $offreg = 2;
   goto GAsBinOffDone;
 end
GAsBinOffWideScratch:
 call GenSetRegFromOperand($scratchreg, $off);
$offreg = 1;
GAsBinOffDone:
 if $offreg is zero:
  $tmp = call NeedsWideOperandLoad($off);
   if $tmp is nonzero:
     goto GAsBinOffWide;
   end
 end
$tmp = &StrIndent8;
 call AppendCStr($tmp);
 call GenBinopMnemonic($op, $flag);
$tmp = &StrFourSpace;
 call AppendCStr($tmp);
 call GenRegRef($dst);
 call AppendByte(',');
 call GenOperand($base);
 call AppendByte(',');
 if $offreg is nonzero:
  $tmp = $offreg - 2;
   if $tmp is zero:
     call GenRegRef($dst);
```

```
        call AppendByte('\n');
        return;
      end
      call GenRegNum($scratchreg);
      call AppendByte('\n');
      return;
    end
    call GenOperand($off);
    call AppendByte('\n');
    return;
  end
```

GenLoadImm emits an immediate load into register node `$dst`. Small nonnegative values use `SETL`; all other 64-bit values use `SETH` + `ORMH` + `ORML` + `ORL` with hexadecimal wyde immediates. GenLoadImmRegNum uses the same emitter, differing only in how the destination register is printed.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenLoadImmTarget($dst, $reg, $use_regnum):
  if $use_regnum is nonzero:
    call GenRegNum($reg);
    return;
  end
  call GenRegRef($dst);
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def GenLoadImmCore($dst, $reg, $use_regnum, $v):
  register $tmp, $w;
  if $v is neg:
   $tmp = -$v;


   $w = $tmp - 256;
    if $w is neg:
     $w = $tmp;
     $tmp = &StrIndent8;
      call AppendCStr($tmp);
      call AppendByte('N');
      call AppendByte('E');
      call AppendByte('G');
      call AppendByte('U');
      call AppendByte(' ');
      call AppendByte(' ');
      call AppendByte(' ');
      call AppendByte(' ');
      call AppendByte(' ');
      call GenLoadImmTarget($dst, $reg, $use_regnum);
      call AppendByte(',');
      call AppendByte('0');
      call AppendByte(',');
      call AppendInt($w);
      call AppendByte('\n');
      return;
    end
  end
```

```
  if $v is nonneg:
   $tmp = $v - 65536;
    if $tmp is neg:
     $tmp = &StrOpSETL;
      call AppendCStr($tmp);
      call GenLoadImmTarget($dst, $reg, $use_regnum);
      call AppendByte(',');
      call AppendHexWyde($v);
      call AppendByte('\n');
      return;
    end
  end
 $tmp = &StrIndent8;
  call AppendCStr($tmp);
 $tmp = &StrOpSETH;
  call AppendCStr($tmp);
  call GenLoadImmTarget($dst, $reg, $use_regnum);
  call AppendByte(',');
 $w = $v >>u 48;
 $w = $w & 65535;
  call AppendHexWyde($w);
  call AppendByte('\n');

 $tmp = &StrIndent8;
  call AppendCStr($tmp);
 $tmp = &StrOpORMH;
  call AppendCStr($tmp);
  call GenLoadImmTarget($dst, $reg, $use_regnum);
  call AppendByte(',');
 $w = $v >>u 32;
 $w = $w & 65535;
  call AppendHexWyde($w);
  call AppendByte('\n');

 $tmp = &StrIndent8;
  call AppendCStr($tmp);
 $tmp = &StrOpORML;
  call AppendCStr($tmp);
  call GenLoadImmTarget($dst, $reg, $use_regnum);
  call AppendByte(',');
 $w = $v >>u 16;
 $w = $w & 65535;
  call AppendHexWyde($w);
  call AppendByte('\n');

 $tmp = &StrIndent8;
  call AppendCStr($tmp);
 $tmp = &StrOpORL;
  call AppendCStr($tmp);
  call GenLoadImmTarget($dst, $reg, $use_regnum);
  call AppendByte(',');
 $w = $v & 65535;
  call AppendHexWyde($w);
  call AppendByte('\n');
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def GenLoadImm($dst, $v):
  call GenLoadImmCore($dst, 0, 0, $v);
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def GenLoadImmRegNum($reg, $v):
  call GenLoadImmCore(0, $reg, 1, $v);
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def NeedsWideOperandLoad($node) -> $r:
  register $kind, $tmp, $v, $inner;
 $r = 0;
 $kind =  call NodeGet($node, ND_KIND);

 $tmp = $kind - NK_IMM;
  goto NWOLImm  if $tmp  is  zero;
 $tmp = $kind - NK_UNOP;
  goto NWOLUn  if $tmp  is  zero;
  return $r;
NWOLImm:
 $v =  call NodeGet($node, ND_A);
  if $v  is  neg:
    return 1;
  end
 $tmp = $v - 256;
  if $tmp  is  nonneg:
   $r = 1;
  end
  return $r;
NWOLUn:
 $tmp =  call NodeGet($node, ND_C);
 $tmp = $tmp - TK_MINUS;
  if $tmp  is  nonzero:
    return $r;
  end
 $inner =  call NodeGet($node, ND_A);
 $kind =  call NodeGet($inner, ND_KIND);
 $tmp = $kind - NK_IMM;
  if $tmp  is  nonzero:
    return $r;
  end
  return 1;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def GenSetRegFromOperand($reg, $node):
  register $tmp, $kind, $off, $len, $v, $inner;
 $kind = call NodeGet($node, ND_KIND);
 $tmp = $kind - NK_LABELREF;


 goto GSRFOpMaybeImm if $tmp is nonzero;
 $off = call NodeGet($node, ND_A);
 $len = call NodeGet($node, ND_B);
 $tmp = call GenLoadConstNameRegNum($reg, $off, $len);
 if $tmp is nonzero:
   return;
 end
GSRFOpMaybeImm:
 $tmp = call NeedsWideOperandLoad($node);
 if $tmp is zero:
   goto GSRFOpSet;
 end
 $tmp = $kind - NK_IMM;
 goto GSRFOpNegImm if $tmp is nonzero;
 $v = call NodeGet($node, ND_A);
 call GenLoadImmRegNum($reg, $v);
 return;
GSRFOpNegImm:
 $inner = call NodeGet($node, ND_A);
 $v = call NodeGet($inner, ND_A);
 $v = -$v;
 call GenLoadImmRegNum($reg, $v);
 return;
GSRFOpSet:
 $tmp = &StrOpSET;
 call AppendCStr($tmp);
 call GenRegNum($reg);
 call AppendByte(',');
 call GenOperand($node);
 call AppendByte('\n');
 return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def GenSetTargetFromOperand($dst, $node):
  register $tmp, $kind, $off, $len, $sreg;
 $kind = call NodeGet($dst, ND_KIND);
 $tmp = $kind - NK_SREG;


 goto GSTFOpMaybeGet if $tmp is nonzero;
 $tmp = &StrIndent8;
 call AppendCStr($tmp);
 call AppendByte('P');
 call AppendByte('U');
 call AppendByte('T');
 call AppendByte(' ');
 call AppendByte(' ');
 call AppendByte(' ');
 call AppendByte(' ');
 call AppendByte(' ');
 $sreg = call NodeGet($dst, ND_A);
```

```
    call GenEmitSregName($sreg);
    call AppendByte(',');
    call GenOperand($node);
    call AppendByte('\n');
    return;
  GSTFOpMaybeGet:
  $kind = call NodeGet($node, ND_KIND);
  $tmp = $kind - NK_SREG;
    goto GSTFOpMaybeConst if $tmp is nonzero;
  $tmp = &StrIndent8;
    call AppendCStr($tmp);
    call AppendByte('G');
    call AppendByte('E');
    call AppendByte('T');
    call AppendByte(' ');
    call AppendByte(' ');
    call AppendByte(' ');
    call AppendByte(' ');
    call AppendByte(' ');
    call GenRegRef($dst);
    call AppendByte(',');
  $sreg = call NodeGet($node, ND_A);
    call GenEmitSregName($sreg);
    call AppendByte('\n');
    return;
  GSTFOpMaybeConst:
  $tmp = $kind - NK_LABELREF;
    goto GSTFOpSet if $tmp is nonzero;
  $off = call NodeGet($node, ND_A);
  $len = call NodeGet($node, ND_B);
  $tmp = call GenLoadConstName($dst, $off, $len);
    if $tmp is nonzero:
      return;
    end
  GSTFOpSet:
  $tmp = &StrOpSET;
    call AppendCStr($tmp);
    call GenRegRef($dst);
    call AppendByte(',');
    call GenOperand($node);
    call AppendByte('\n');
    return;
  end
```

⟨ ***Generator procedures*** ⟩ +≡

```
  def GenLoadConstName($dst, $off, $len) -> $r:
    register $tmp;
  $r = 0;
  $tmp = call LookupBuiltinImm($off, $len);
    if $tmp is zero:
      goto GLCConst;
    end
  $tmp = $tmp - 1;
    call GenLoadImm($dst, $tmp);
    return 1;
  GLCConst:
  $tmp = call LookupConstExpr($off, $len);
    if $tmp is zero:
      return $r;
```

```
    end
  $tmp = call IsConstExprNode($tmp);
   if $tmp is zero:
     return $r;
   end
  $tmp = call LookupConstExpr($off, $len);
  $tmp = call EvalConstExpr($tmp);
   call GenLoadImm($dst, $tmp);
   return 1;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def GenLoadConstNameRegNum($reg, $off, $len) -> $r:
   register $tmp;
  $r = 0;
  $tmp = call LookupBuiltinImm($off, $len);
   if $tmp is zero:
     goto GLCNConst;
   end
  $tmp = $tmp - 1;
   call GenLoadImmRegNum($reg, $tmp);
   return 1;
 GLCNConst:
  $tmp = call LookupConstExpr($off, $len);
   if $tmp is zero:
     return $r;
   end
  $tmp = call IsConstExprNode($tmp);
   if $tmp is zero:
     return $r;
   end
  $tmp = call LookupConstExpr($off, $len);
  $tmp = call EvalConstExpr($tmp);
   call GenLoadImmRegNum($reg, $tmp);
   return 1;
end
```

`GenUnopAssign` emits `$dst = UNOP operand;`. Three token-encoded forms are supported when `ND_D`= 0: `TK_MINUS` maps to `NEG $dst,0,$src`; `TK_MINUSU` maps to `NEGU $dst,0,$src`. A nonzero `ND_D` with `ND_C` in `1`…`8` is one of the eight float/conversion unops `sqrt`, `int`, `flot`, `flotu`, `fix`, `fixu`, `sflot`, `sflotu`, which assemble as `FSQRT`/`FINT`/`FLOT`/`FLOTU`/`FIX`/`FIXU`/`SFLOT`/`SFLOTU`. The bracketed rounding mode saved in `extra[0]` is currently ignored—MMIXAL will default to `rA`'s mode.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenUnopAssign($dst, $rhs):
  register $tmp, $op, $flag, $src, $src_kind;
 $op   =  call NodeGet($rhs, ND_C);
 $flag =  call NodeGet($rhs, ND_D);


 $src  =  call NodeGet($rhs, ND_A);
 $tmp = &StrIndent8;
  call AppendCStr($tmp);
  if $flag  is  zero:
   $tmp = $op - TK_MINUS;
    if $tmp  is  zero:
     $src_kind =  call NodeGet($src, ND_KIND);
     $tmp = $src_kind - NK_IMM;
      if $tmp  is  zero:
       $tmp =  call NodeGet($src, ND_A);
       $tmp = -$tmp;
        call GenLoadImm($dst, $tmp);
        return;
      end
     $tmp = &StrOpNEG;
      call AppendCStr($tmp);
      call GenRegRef($dst);
      call AppendByte(',');
      call AppendByte('0');
      call AppendByte(',');
      call GenOperand($src);
      call AppendByte('\n');
      return;
    end
   $tmp = $op - TK_MINUSU;
    if $tmp  is  zero:
     $tmp = &StrOpNEGU;
      call AppendCStr($tmp);
      call GenRegRef($dst);
      call AppendByte(',');
      call AppendByte('0');
      call AppendByte(',');
      call GenOperand($src);
      call AppendByte('\n');
      return;
    end
    call AppendByte('?');
    call AppendByte('?');
    call AppendByte('?');
    call AppendByte('\n');
    return;
  end
$tmp = $op - 1;
 if $tmp  is  zero: $tmp = &StrOpFSQRT;   call AppendCStr($tmp);  goto GUADst;  end
$tmp = $op - 2;
 if $tmp  is  zero: $tmp = &StrOpFINT;    call AppendCStr($tmp);  goto GUADst;  end
$tmp = $op - 3;
 if $tmp  is  zero: $tmp = &StrOpFLOT;    call AppendCStr($tmp);  goto GUADst;  end
$tmp = $op - 4;
 if $tmp  is  zero: $tmp = &StrOpFLOTU;   call AppendCStr($tmp);  goto GUADst;  end
$tmp = $op - 5;
 if $tmp  is  zero: $tmp = &StrOpFIX;     call AppendCStr($tmp);  goto GUADst;  end
$tmp = $op - 6;
 if $tmp  is  zero: $tmp = &StrOpFIXU;    call AppendCStr($tmp);  goto GUADst;  end
$tmp = $op - 7;
```

```
    if $tmp is zero: $tmp = &StrOpSFLOT;   call AppendCStr($tmp);  goto GUADst;  end
   $tmp = &StrOpSFLOTU;
    call AppendCStr($tmp);
   GUADst:
    call GenRegRef($dst);
    call AppendByte(',');
    call GenOperand($src);
    call AppendByte('\n');
    return;
  end
```

`GenStore` emits `STx $src,$base,$off` for an `NK_STORE` node. `ND_A`=base, `ND_B`=offset (zero when absent), `ND_C`=size code, `ND_D`=RHS. The RHS must be an atom—the parser does not accept a binary expression in store position, and a compound RHS would need a spill temporary that the bootstrap does not yet allocate.

⟨ ***Generator procedures*** ⟩ +≡

```
  def GenStore($node):
    register $tmp, $base, $off, $sz, $rhs, $kind, $nameoff, $namelen;
   $base =  call NodeGet($node, ND_A);
   $off  =  call NodeGet($node, ND_B);


   $sz   =  call NodeGet($node, ND_C);
   $rhs  =  call NodeGet($node, ND_D);
   $kind =  call NodeGet($rhs, ND_KIND);
   $tmp = $kind - NK_REG;
    goto GStHaveSrc  if $tmp  is  zero;
   $tmp = $kind - NK_REGNAME;
    goto GStCheckName  if $tmp  is  zero;
   $tmp = $kind - NK_SREG;
    goto GStHaveSrc  if $tmp  is  zero;
    goto GStNeedTmp;
   GStCheckName:
   $nameoff =  call NodeGet($rhs, ND_A);
   $namelen =  call NodeGet($rhs, ND_B);
   $tmp =  call LookupAlias($nameoff, $namelen);
    goto GStHaveSrc  if $tmp  is  nonzero;
   $tmp =  call IsGregName($nameoff, $namelen);
    goto GStHaveSrc  if $tmp  is  nonzero;
   GStNeedTmp:
   $tmp = $kind - NK_LABELREF;
    goto GStNeedTmpConst  if $tmp  is  zero;
    goto GStNeedTmpEmit;
   GStNeedTmpConst:
   $nameoff =  call NodeGet($rhs, ND_A);
   $namelen =  call NodeGet($rhs, ND_B);
   $tmp =  call GenLoadConstNameRegNum(255, $nameoff, $namelen);
    if $tmp  is  nonzero:
      goto GStHaveSrc;
    end
   GStNeedTmpEmit:
    call GenSetRegFromOperand(255, $rhs);
   GStHaveSrc:
   $tmp = &StrIndent8;
    call AppendCStr($tmp);
    call GenSizeOp($sz, 1);
   $tmp = $kind - NK_REG;
    goto GStSrcNode  if $tmp  is  zero;
```

```
  $tmp = $kind - NK_REGNAME;
   goto GStSrcName  if $tmp  is  zero;
  $tmp = $kind - NK_SREG;
   goto GStSrcNode  if $tmp  is  zero;
   call AppendByte('$');
   call AppendInt(255);
   goto GStSrcDone;
  GStSrcName:
  $nameoff =  call NodeGet($rhs, ND_A);
  $namelen =  call NodeGet($rhs, ND_B);
  $tmp =  call LookupAlias($nameoff, $namelen);
   goto GStSrcNode  if $tmp  is  nonzero;
  $tmp =  call IsGregName($nameoff, $namelen);
   goto GStSrcNode  if $tmp  is  nonzero;
   call AppendByte('$');
   call AppendInt(255);
   goto GStSrcDone;
  GStSrcNode:
   call GenOperand($rhs);
  GStSrcDone:
   call AppendByte(',');
   call GenOperand($base);
   call AppendByte(',');
   if $off  is  nonzero:
     call GenOperand($off);
   end
   if $off  is  zero:
     call AppendByte('0');
   end
   call AppendByte('\n');
   return;
 end
```

`GenCall` emits a call statement for both `NK_CALL` (`$isAsn`= 0) and `NK_CALLASN` (`$isAsn`= 1). When inside a procedure, it preserves the return address (`$rJ`) and uses a procedure-local call base (4).*Attoplevel*, *ituses*255 directly.

The generated sequence is:

- • for each argument, a `! arg[i] = <operand>` comment;
- • (in procedures only) `GET $2, rJ` to save the return address;
- • a `PUSHJ` (or `PUSHGO`) with context-appropriate call base;
- • for `NK_CALLASN`, move return values from call-base registers to destinations;
- • (in procedures only) `PUT rJ, $2` to restore the return address;
- • (for comments) a `! returns: $dst, ...` comment list.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenCall($node, $isAsn):
  register $tmp, $callee, $args, $arg, $idx, $indirect, $dsts, $d, $first, $callbase, $retidx, $argcnt,
    $tempbase, $regnum;
 $dsts     =  call NodeGet($node, ND_A);
 $callee   =  call NodeGet($node, ND_B);


 $args     =  call NodeGet($node, ND_C);
 $indirect =  call NodeGet($node, ND_D);
  # Pick a call-window base above the highest-allocated caller register so
  # PUSHJ's saved range covers the entire live local frame.  Without this,
  # functions whose register usage grows past the historical default of 19
  # have their locals silently clobbered by the callee's frame.
 $callbase = 19;
 $tmp = $GlobalRegUsed;
  if $gen_in_proc  is  nonzero:
   $tmp = $LocalRegUsed;
  end
 $tmp =  call MaxRegUsed($tmp);
 $tmp = $tmp + 1;
  if $tmp >u $callbase:
   $callbase = $tmp;
  end
 $last_callbase = $callbase;
  # Argument-setup commentary.
 $arg = $args;
 $idx = 0;
 $argcnt = 0;
 GClArg:
  if $arg  is  zero:
    goto GClArgDone;
  end
 $tmp = &StrCallArg;
  call AppendCStr($tmp);
  call AppendInt($idx);
 $tmp = &StrCallArgEq;
  call AppendCStr($tmp);
  call GenOperand($arg);
  call AppendByte('\n');
 $argcnt = $argcnt + 1;
 $arg =  call NodeGet($arg, ND_NEXT);
 $idx = $idx + 1;
  goto GClArg;
 GClArgDone:
  # Materialize arguments into temporaries, then copy into the call window.
 $tempbase = $callbase + $argcnt;
 $tempbase = $tempbase + 1;
 $arg = $args;
 $idx = 0;
 GClSaveArg:
  if $arg  is  zero:
    goto GClLoadArg;
  end
 $regnum = $tempbase + $idx;
 $tmp =  call NodeGet($arg, ND_KIND);
 $d = $tmp - NK_LABELREF;
  if $d  is  nonzero:
    goto GClSaveArgSet;
  end
 $tmp =  call NodeGet($arg, ND_A);
```

```
$d =  call NodeGet($arg, ND_B);
$first =  call GenLoadConstNameRegNum($regnum, $tmp, $d);
 if $first  is  nonzero:
   goto GClSaveArgNext;
 end
GClSaveArgSet:
 call GenSetRegFromOperand($regnum, $arg);
GClSaveArgNext:
$arg =  call NodeGet($arg, ND_NEXT);
$idx = $idx + 1;
 goto GClSaveArg;
GClLoadArg:
$idx = 0;
GClLoadArgLoop:
$tmp = $idx - $argcnt;
 if $tmp  is  nonneg:
   goto GClSaveRJ;
 end
$tmp = &StrOpSET;
 call AppendCStr($tmp);
$regnum = $callbase + $idx;
$regnum = $regnum + 1;
 call GenRegNum($regnum);
 call AppendByte(',');
$regnum = $tempbase + $idx;
 call GenRegNum($regnum);
 call AppendByte('\n');
$idx = $idx + 1;
 goto GClLoadArgLoop;
GClSaveRJ:
 # Save rJ when inside a procedure body.
 if $gen_in_proc  is  nonzero:
  $tmp = &StrGetRJ;
   call AppendCStr($tmp);
 end
GClJump:
 if $indirect  is  nonzero:
  $tmp = &StrOpPUSHGO;
   call AppendCStr($tmp);
   call AppendInt($callbase);
   call AppendByte(',');
   call GenRegRef($callee);
  $tmp = &StrCallZero;
   call AppendCStr($tmp);
   goto GClMoveRets;
 end
$tmp = &StrOpPUSHJ;
 call AppendCStr($tmp);
 call AppendInt($callbase);
 call AppendByte(',');
 call GenLabelName($callee);
 call AppendByte('\n');
GClMoveRets:
 # Move return values from call-base registers to destination registers.
 if $isAsn  is  nonzero:
  $d = $dsts;
  $retidx = 0;
  GClMoveRet:
   if $d  is  zero:
     goto GClRetCommentStart;
   end
  $tmp = &StrOpSET;
```

```
    call AppendCStr($tmp);
    call GenRegRef($d);
    call AppendByte(',');
    call AppendByte('$');
   $tmp = $callbase + $retidx;
    call AppendInt($tmp);
    call AppendByte('\n');
   $retidx = $retidx + 1;
   $d =  call NodeGet($d, ND_NEXT);
    goto GClMoveRet;
  end
 GClRetCommentStart:
  # Return-register list commentary (call-assign only).
  if $isAsn  is  nonzero:
   $tmp = &StrCallRet;
    call AppendCStr($tmp);
   $d = $dsts;
   $first = 1;
   GClRetComment:
    if $d  is  zero:
      call AppendByte('\n');
      goto GClRestore;
    end
    if $first  is  zero:
      call AppendByte(',');
      call AppendByte(' ');
    end
   $first = 0;
    call GenRegRef($d);
   $d =  call NodeGet($d, ND_NEXT);
    goto GClRetComment;
  end
 GClRestore:
  # Restore rJ when inside a procedure body.
  if $gen_in_proc  is  nonzero:
   $tmp = &StrPutRJ;
    call AppendCStr($tmp);
  end
 GClDone:
  return;
end
```

26. Generator: control flow.

Loops and conditionals introduce generator-synthesised labels of the form `_L<n>`, where `<n>` is a monotonic counter held in the global `$label_seq`. `GenNewLabel` allocates a fresh ID (starting at 1 so that a zero return from `NodeGet` remains distinguishable). `GenEmitLabelRef` writes `_L<n>` to `OutputBuf`, and `GenEmitLabelDef` writes `_L<n>\\tIS\\t@\\n`, pinning the label to the current location counter.

Break/continue targets are tracked in two global registers (`$BreakLbl`, `$ContLbl`) carrying label IDs or zero for "no enclosing loop". Because each `GenWhile`/`GenLoop`/`GenDo` saves the incoming values into proc-local registers before overwriting the globals and restores them on the way out, nested loops work naturally—the `PUSHJ` at every helper call preserves the caller's local copies.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenNewLabel() -> $r:
 $label_seq = $label_seq + 1;
  return $label_seq;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def GenEmitLabelRef($id):
  call AppendByte('_');
  call AppendByte('L');
  call AppendInt($id);
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def GenEmitLabelDef($id):
  register $tmp;
  call GenEmitLabelRef($id);
 $tmp = &StrIsHere;
  call AppendCStr($tmp);
  return;
end
```

`GenJmpTo` writes `JMP _L<id>\n`—an unconditional branch to a synthesised label.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenJmpTo($id):
  register $tmp;
 $tmp = &StrJmpDir;
  call AppendCStr($tmp);
  call GenEmitLabelRef($id);
  call AppendByte('\n');
  return;
end
```

`GenPolarityMnemonic` writes the 8-wide MMIX branch mnemonic for the given polarity keyword code. When `$invert` $\neq 0$ the complementary test is used, so `is zero` inverts to `BNZ`, `is neg` to `BNN`, and so on. Non-polarity codes (including relational ones) emit `???` so the stream remains parseable.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenPolarityMnemonic($pol, $invert):
  register $code, $tmp;
 $code = $pol;
  if $invert  is  nonzero:


    $tmp = $pol - KW_ZERO;
     if $tmp  is  zero:    $code = KW_NONZERO;  goto GPMPick;  end
    $tmp = $pol - KW_NONZERO;
     if $tmp  is  zero:    $code = KW_ZERO;     goto GPMPick;  end
    $tmp = $pol - KW_NEG;
     if $tmp  is  zero:    $code = KW_NONNEG;   goto GPMPick;  end
    $tmp = $pol - KW_NONNEG;
     if $tmp  is  zero:    $code = KW_NEG;      goto GPMPick;  end
    $tmp = $pol - KW_POS;
     if $tmp  is  zero:    $code = KW_NONPOS;   goto GPMPick;  end
    $tmp = $pol - KW_NONPOS;
     if $tmp  is  zero:    $code = KW_POS;      goto GPMPick;  end
    $tmp = $pol - KW_ODD;
     if $tmp  is  zero:    $code = KW_EVEN;     goto GPMPick;  end
    $tmp = $pol - KW_EVEN;
     if $tmp  is  zero:    $code = KW_ODD;      goto GPMPick;  end
  end
 GPMPick:
 $tmp = $code - KW_ZERO;
  if $tmp  is  zero: $tmp = &StrBZ;   call AppendCStr($tmp);  return;  end
 $tmp = $code - KW_NONZERO;
  if $tmp  is  zero: $tmp = &StrBNZ;  call AppendCStr($tmp);  return;  end
 $tmp = $code - KW_NEG;
  if $tmp  is  zero: $tmp = &StrBN;   call AppendCStr($tmp);  return;  end
 $tmp = $code - KW_NONNEG;
  if $tmp  is  zero: $tmp = &StrBNN;  call AppendCStr($tmp);  return;  end
 $tmp = $code - KW_POS;
  if $tmp  is  zero: $tmp = &StrBP;   call AppendCStr($tmp);  return;  end
 $tmp = $code - KW_NONPOS;
  if $tmp  is  zero: $tmp = &StrBNP;  call AppendCStr($tmp);  return;  end
 $tmp = $code - KW_ODD;
  if $tmp  is  zero: $tmp = &StrBOD;  call AppendCStr($tmp);  return;  end
 $tmp = $code - KW_EVEN;
  if $tmp  is  zero: $tmp = &StrBEV;  call AppendCStr($tmp);  return;  end
  call AppendByte('?');
  call AppendByte('?');
  call AppendByte('?');
  return;
end
```

`GenCondBranch` emits a branch sequence that jumps to label ID `$target` when the `NK_COND` argument evaluates to true (`$invert`= 0) or false (`$invert`= 1). Two condition shapes are supported: polarity tests (`ND_D`= 0) render as a single `Bxx` on the tested register, while relational comparisons (`ND_D`≠ 0) emit a `CMP` (or `CMPU` / `FCMP`) into scratch register `$255` followed by a polarity-flavoured branch on that scratch.

`NK_COND` layout: `ND_A` = register-reference node (LHS), `ND_C` = polarity keyword code for polarity form or a `TK_EQ`…`TK_GEQF` token code for relop form, `ND_D` = 0 for polarity, else the RHS expression of the comparison.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenCondBranch($cond, $target, $invert):
  register $tmp, $reg, $code, $rhs, $k;
 $rhs  =  call NodeGet($cond, ND_D);
 $reg  =  call NodeGet($cond, ND_A);


 $code =  call NodeGet($cond, ND_C);
  if $rhs  is  zero:
   $tmp = &StrIndent8;
    call AppendCStr($tmp);
    call GenPolarityMnemonic($code, $invert);
    call GenRegRef($reg);
    call AppendByte(',');
    call GenEmitLabelRef($target);
    call AppendByte('\n');
    return;
  end
 $tmp = &StrIndent8;
  call AppendCStr($tmp);
 $k = $code - TK_EQF;
  if $k  is  nonneg:
   $tmp = &StrOpFCMP;
    call AppendCStr($tmp);
    goto GCBHave;
  end
 $k = $code - TK_LTU;
  if $k  is  nonneg:
   $tmp = &StrOpCMPU;
    call AppendCStr($tmp);
    goto GCBHave;
  end
 $tmp = &StrOpCMP;
  call AppendCStr($tmp);
GCBHave:
  call AppendByte('$');
  call AppendByte('2');
  call AppendByte('5');
  call AppendByte('5');
  call AppendByte(',');
  call GenRegRef($reg);
  call AppendByte(',');
  call GenOperand($rhs);
  call AppendByte('\n');
 $tmp = &StrIndent8;
  call AppendCStr($tmp);
  call GenRelopBranchMnem($code, $invert);
  call AppendByte('$');
  call AppendByte('2');
  call AppendByte('5');
  call AppendByte('5');
  call AppendByte(',');
  call GenEmitLabelRef($target);
  call AppendByte('\n');
  return;
end
```

GenRelopBranchMnem writes the Bxx mnemonic that follows a CMP for a relational comparison. The six comparison families (eq, ne, lt, le, gt, ge, each appearing in signed, unsigned, and

float flavours) map onto the branch polarities `BZ`/`BNZ`/`BN`/`BNP`/`BP`/`BNN` respectively, and `$invert` swaps each with its complement.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenRelopBranchMnem($code, $invert):
  register $tmp, $k;
 $k = $code - TK_EQ;
  goto GRBEQ  if $k  is  zero;


 $k = $code - TK_EQF;
  goto GRBEQ  if $k  is  zero;
 $k = $code - TK_NEQ;
  goto GRBNE  if $k  is  zero;
 $k = $code - TK_NEQF;
  goto GRBNE  if $k  is  zero;
 $k = $code - TK_LT;
  goto GRBLT  if $k  is  zero;
 $k = $code - TK_LTU;
  goto GRBLT  if $k  is  zero;
 $k = $code - TK_LTF;
  goto GRBLT  if $k  is  zero;
 $k = $code - TK_LEQ;
  goto GRBLE  if $k  is  zero;
 $k = $code - TK_LEQU;
  goto GRBLE  if $k  is  zero;
 $k = $code - TK_LEQF;
  goto GRBLE  if $k  is  zero;
 $k = $code - TK_GT;
  goto GRBGT  if $k  is  zero;
 $k = $code - TK_GTU;
  goto GRBGT  if $k  is  zero;
 $k = $code - TK_GTF;
  goto GRBGT  if $k  is  zero;
  goto GRBGE;
 GRBEQ:
  if $invert  is  zero: $tmp = &StrBZ;   call AppendCStr($tmp);  return;  end
 $tmp = &StrBNZ;  call AppendCStr($tmp);  return;
 GRBNE:
  if $invert  is  zero: $tmp = &StrBNZ;  call AppendCStr($tmp);  return;  end
 $tmp = &StrBZ;   call AppendCStr($tmp);  return;
 GRBLT:
  if $invert  is  zero: $tmp = &StrBN;   call AppendCStr($tmp);  return;  end
 $tmp = &StrBNN;  call AppendCStr($tmp);  return;
 GRBLE:
  if $invert  is  zero: $tmp = &StrBNP;  call AppendCStr($tmp);  return;  end
 $tmp = &StrBP;   call AppendCStr($tmp);  return;
 GRBGT:
  if $invert  is  zero: $tmp = &StrBP;   call AppendCStr($tmp);  return;  end
 $tmp = &StrBNP;  call AppendCStr($tmp);  return;
 GRBGE:
  if $invert  is  zero: $tmp = &StrBNN;  call AppendCStr($tmp);  return;  end
 $tmp = &StrBN;   call AppendCStr($tmp);  return;
end
```

`GenWhile` implements `while cond:  body end`. Layout emitted:

```
exttt_Ltop IS @
exttt <cond-false jump to _Lend>
```

```
    exttt body
    exttt JMP _Ltop
    exttt_Lend IS @
```

with `$BreakLbl`/`$ContLbl` bound to `_Lend`/`_Ltop` across the body so that nested `break`/`continue` statements resolve correctly.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenWhile($node):
  register $prevBrk, $prevCont, $top, $end, $cond, $body;
 $prevBrk  = $BreakLbl;
 $prevCont = $ContLbl;
 $top =  call GenNewLabel();
 $end =  call GenNewLabel();
 $BreakLbl = $end;
 $ContLbl  = $top;
  call GenEmitLabelDef($top);
 $cond =  call NodeGet($node, ND_A);
  call GenCondBranch($cond, $end, 1);
 $body =  call NodeGet($node, ND_B);
  call GenStmtList($body);
  call GenJmpTo($top);
  call GenEmitLabelDef($end);
 $BreakLbl = $prevBrk;
 $ContLbl  = $prevCont;
  return;
end
```

`GenLoop` implements the infinite loop `loop: body end`. Same structure as `GenWhile` minus the head test. `GenDo` emits an identical skeleton for now.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenLoop($node):
  register $prevBrk, $prevCont, $top, $end, $body;
 $prevBrk  = $BreakLbl;
 $prevCont = $ContLbl;
 $top =  call GenNewLabel();
 $end =  call GenNewLabel();
 $BreakLbl = $end;
 $ContLbl  = $top;
  call GenEmitLabelDef($top);
 $body =  call NodeGet($node, ND_A);
  call GenStmtList($body);
  call GenJmpTo($top);
  call GenEmitLabelDef($end);
 $BreakLbl = $prevBrk;
 $ContLbl  = $prevCont;
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def GenDo($node):
  register $cond, $prevBrk, $prevCont, $top, $cont, $end, $body;
 $cond = call NodeGet($node, ND_B);
  if $cond is zero:
    call GenLoop($node);
    return;
  end
 $prevBrk  = $BreakLbl;
 $prevCont = $ContLbl;
 $top  = call GenNewLabel();
 $cont = call GenNewLabel();
 $end  = call GenNewLabel();
 $BreakLbl = $end;
 $ContLbl  = $cont;
  call GenEmitLabelDef($top);
 $body = call NodeGet($node, ND_A);
  call GenStmtList($body);
  call GenEmitLabelDef($cont);
  call GenCondBranch($cond, $top, 0);
  call GenEmitLabelDef($end);
 $BreakLbl = $prevBrk;
 $ContLbl  = $prevCont;
  return;
end
```

`GenIf` implements |if cond: body [elif cond: body]* [else body] end. `All arms--the leading` if `and every` elif|—share a common shape: “branch to next arm on false, emit body, jump to `_Lend`, plant next-arm label.” `GenIfArm` encapsulates this so the walker can reuse it for the initial condition and for every link in the `NK_ELIF` chain that hangs off `ND_C`.

Output shape for three-arm `if/elif/elif/else`:

```
exttt <cond_0-false jump to _L1>
exttt body_0
exttt JMP _Lend
exttt_L1 IS @
exttt <cond_1-false jump to _L2>
exttt body_1
exttt JMP _Lend
exttt_L2 IS @
exttt <cond_2-false jump to _Lelse>
exttt body_2
exttt JMP _Lend
exttt_Lelse IS @
exttt else-body
exttt_Lend IS @
```

If there is no `else` branch the final arm’s “next” target collapses to `_Lend` and the trailing `JMP _Lend` is dropped.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenIfArm($cond, $body, $restElifs, $elseBody, $lEnd):
  register $lNext;
  if $restElifs is zero:
    if $elseBody is zero:
      call GenCondBranch($cond, $lEnd, 1);
      call GenStmtList($body);
      return;
    end
  end
 $lNext = call GenNewLabel();
  call GenCondBranch($cond, $lNext, 1);
  call GenStmtList($body);
  call GenJmpTo($lEnd);
  call GenEmitLabelDef($lNext);
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def GenIf($node):
  register $cond, $body, $elifHead, $elseBody, $lEnd, $elif, $nextE, $c, $b;
 $cond     = call NodeGet($node, ND_A);
 $body     = call NodeGet($node, ND_B);
 $elifHead = call NodeGet($node, ND_C);
 $elseBody = call NodeGet($node, ND_D);
 $lEnd = call GenNewLabel();
  call GenIfArm($cond, $body, $elifHead, $elseBody, $lEnd);
 $elif = $elifHead;
 GIfElLp:
  goto GIfElDn if $elif is zero;
 $nextE = call NodeGet($elif, ND_NEXT);
 $c = call NodeGet($elif, ND_A);
 $b = call NodeGet($elif, ND_B);
  call GenIfArm($c, $b, $nextE, $elseBody, $lEnd);
 $elif = $nextE;
  goto GIfElLp;
 GIfElDn:
  if $elseBody is nonzero:
    call GenStmtList($elseBody);
  end
  call GenEmitLabelDef($lEnd);
  return;
end
```

`GenBreak` and `GenContinue` emit a `JMP` to the target label recorded in the currently active `$BreakLbl`/`$ContLbl`. A zero value means "no enclosing loop," which is a static error in the source language but is rendered here as a self-describing comment so the surrounding output is still legal MMIXAL. A conditional tail (`ND_C`$\neq 0$, produced by the parser's `IfCondTail`) flips the branch into the polarity-aware form.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenBreak($node):
  register $cond, $tmp;
  if $BreakLbl is zero:
   $tmp = &StrBrkOut;
    call AppendCStr($tmp);
    call AppendByte('\n');
    return;
  end
 $cond = call NodeGet($node, ND_C);
  if $cond is zero:
    call GenJmpTo($BreakLbl);
    return;
  end
  call GenCondBranch($cond, $BreakLbl, 0);
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def GenContinue($node):
  register $cond, $tmp;
  if $ContLbl is zero:
   $tmp = &StrContOut;
    call AppendCStr($tmp);
    call AppendByte('\n');
    return;
  end
 $cond = call NodeGet($node, ND_C);
  if $cond is zero:
    call GenJmpTo($ContLbl);
    return;
  end
  call GenCondBranch($cond, $ContLbl, 0);
  return;
end
```

Strings introduced by the control-flow emitters.

⟨ ***Data segment*** ⟩ +≡

```
byte StrBZ        "BZ      ", 0;
byte StrBNZ       "BNZ     ", 0;
byte StrBN        "BN      ", 0;
byte StrBNN       "BNN     ", 0;
byte StrBP        "BP      ", 0;
byte StrBNP       "BNP     ", 0;
byte StrBOD       "BOD     ", 0;
byte StrBEV       "BEV     ", 0;
byte StrOpCMP     "CMP     ", 0;
byte StrOpCMPU    "CMPU    ", 0;
byte StrOpFCMP    "FCMP    ", 0;
byte StrBrkOut    "        ! break outside loop", 0;
byte StrContOut   "        ! continue outside loop", 0;
byte StrRndGet    "        GET     $255,rA\n", 0;
byte StrRndClr    "        ANDNMH  $255,#0003\n", 0;
byte StrRndSetA   "        INCMH   $255,#000", 0;
byte StrRndSetB   "\n", 0;
```

```
byte StrRndPut    "        PUT     rA,$255\n", 0;
```

Strings used by the statement emitters.

⟨***Data segment***⟩ +≡

```
byte StrJmpDir      "        JMP     ", 0;
byte StrGoInd       "        GO      $255,", 0;
byte StrSaveDir     "        SAVE    ", 0;
byte StrUnsaveDir   "        UNSAVE  0,", 0;


byte StrCommaZero   ",0\n", 0;
byte StrOpADD       "ADD", 0;
byte StrOpADDU      "ADDU", 0;
byte StrOpSUB       "SUB", 0;
byte StrOpSUBU      "SUBU", 0;
byte StrOpMUL       "MUL", 0;
byte StrOpMULU      "MULU", 0;
byte StrOpDIV       "DIV", 0;
byte StrOpDIVU      "DIVU", 0;
byte StrOpFADD      "FADD", 0;
byte StrOpFSUB      "FSUB", 0;
byte StrOpFMUL      "FMUL", 0;
byte StrOpFDIV      "FDIV", 0;
byte StrOpFREM      "FREM", 0;
byte StrOpAND       "AND", 0;
byte StrOpOR        "OR", 0;
byte StrOpXOR       "XOR", 0;
byte StrOpANDN      "ANDN", 0;
byte StrOpORN       "ORN", 0;
byte StrOpNAND      "NAND", 0;
byte StrOpNOR       "NOR", 0;
byte StrOpNXOR      "NXOR", 0;
byte StrOpMOR       "MOR", 0;
byte StrOpMXOR      "MXOR", 0;
byte StrOpMUX       "MUX", 0;
byte StrOpSADD      "SADD", 0;
byte StrOpBDIF      "BDIF", 0;
byte StrOpWDIF      "WDIF", 0;
byte StrOpTDIF      "TDIF", 0;
byte StrOpODIF      "ODIF", 0;
byte StrOpFCMPE     "FCMPE", 0;
byte StrOpFEQLE     "FEQLE", 0;
byte StrOpFUNE      "FUNE", 0;
byte StrOpSL        "SL", 0;
byte StrOpSR        "SR", 0;
byte StrOpSLU       "SLU", 0;
byte StrOpSRU       "SRU", 0;
byte StrOp2ADDU     "2ADDU", 0;
byte StrOp4ADDU     "4ADDU", 0;
byte StrOp8ADDU     "8ADDU", 0;
byte StrOp16ADDU    "16ADDU", 0;
byte StrOpSET       "        SET     ", 0;
byte StrOpSETL      "        SETL    ", 0;
byte StrOpGETA      "        GETA    ", 0;
byte StrFourSpace   "     ", 0;
byte StrAsnStub     "        ! assign (kind ", 0;
byte StrCallArg     "        ! arg[", 0;
byte StrCallArgEq   "] = ", 0;
```

```
byte StrCallRet     "        ! returns: ", 0;
byte StrOpPUSHJ     "        PUSHJ   $", 0;
byte StrOpPUSHGO    "        PUSHGO  $", 0;
byte StrCallZero    ",0\n", 0;
byte StrGetRJ       "        GET $17,rJ\n", 0;
byte StrPutRJ       "        PUT rJ,$17\n", 0;
byte StrOpPOP       "POP     ", 0;
byte StrOpNEG       "NEG     ", 0;
byte StrOpNEGU      "NEGU    ", 0;
byte StrOpSETH      "SETH    ", 0;
byte StrOpFSQRT     "FSQRT   ", 0;
byte StrOpFINT      "FINT    ", 0;
byte StrOpFLOT      "FLOT    ", 0;
byte StrOpFLOTU     "FLOTU   ", 0;
byte StrOpFIX       "FIX     ", 0;
byte StrOpFIXU      "FIXU    ", 0;
byte StrOpSFLOT     "SFLOT   ", 0;
byte StrOpSFLOTU    "SFLOTU  ", 0;
byte StrOpANDNH     "ANDNH   ", 0;
byte StrOpANDNL     "ANDNL   ", 0;
byte StrOpANDNMH    "ANDNMH  ", 0;
byte StrOpANDNML    "ANDNML  ", 0;
byte StrOpORH       "ORH     ", 0;
byte StrOpORL       "ORL     ", 0;
byte StrOpORMH      "ORMH    ", 0;
byte StrOpORML      "ORML    ", 0;
```

GenLabelStmt emits `Label IS @` for an NK_LABELSTMT node, pinning the name to the current location counter. MMIXAL accepts this idiom wherever a label declaration would otherwise share a line with an instruction.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenLabelStmt($node):
  register $tmp, $off, $len;
  if $gen_in_proc is nonzero:
   $tmp = call NodeGet($node, ND_C);
    if $tmp is nonzero:
      call GenScopedLabelName($node);
      goto GLSDone;
    end
  end
  call GenLabelName($node);
 GLSDone:
 $tmp = &StrIsHere;
  call AppendCStr($tmp);
  return;
end
```

GenTrap emits `TRAP x,y,z\n` for an NK_TRAP node, using GenConstExpr on each of the three operand subtrees in ND_A/ND_B/ND_C.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenTrap($node):
  register $tmp, $x, $y, $z;
 $tmp = &StrTrapDir;
  call AppendCStr($tmp);
 $x =  call NodeGet($node, ND_A);
 $y =  call NodeGet($node, ND_B);
 $z =  call NodeGet($node, ND_C);
  call GenConstExpr($x);
  call AppendByte(',');
  call GenConstExpr($y);
  call AppendByte(',');
  call GenConstExpr($z);
  call AppendByte('\n');
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def GenAddrYZ($base, $off):
  call GenOperand($base);
  call AppendByte(',');
  if $off  is  zero:
    call AppendByte('0');
    return;
  end
  call GenOperand($off);
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def GenPreldLike($node):
  register $tmp, $kw, $x, $base, $off;
 $kw =  call NodeGet($node, ND_A);
 $x =  call NodeGet($node, ND_B);


 $base =  call NodeGet($node, ND_C);
 $off =  call NodeGet($node, ND_D);
 $tmp = &StrIndent8;
  call AppendCStr($tmp);
 $tmp = $kw - KW_PRELD;
  if $tmp  is  zero:
    call AppendByte('P');  call AppendByte('R');  call AppendByte('E');  call
     AppendByte('L');  call AppendByte('D');
    call AppendByte(' ');  call AppendByte(' ');  call AppendByte(' ');
    goto GPLHaveOp;
  end
 $tmp = $kw - KW_PREGO;
  if $tmp  is  zero:
    call AppendByte('P');  call AppendByte('R');  call AppendByte('E');  call
     AppendByte('G');  call AppendByte('O');
    call AppendByte(' ');  call AppendByte(' ');  call AppendByte(' ');
    goto GPLHaveOp;
  end
 $tmp = $kw - KW_PREST;
  if $tmp  is  zero:
```

```
    call AppendByte('P');  call AppendByte('R');  call AppendByte('E');  call
     AppendByte('S');  call AppendByte('T');
    call AppendByte(' ');  call AppendByte(' ');  call AppendByte(' ');
    goto GPLHaveOp;
  end
 $tmp = $kw - KW_SYNCD;
  if $tmp  is  zero:
    call AppendByte('S');  call AppendByte('Y');  call AppendByte('N');  call
     AppendByte('C');  call AppendByte('D');
    call AppendByte(' ');  call AppendByte(' ');  call AppendByte(' ');
    goto GPLHaveOp;
  end
 $tmp = $kw - KW_SYNCID;
  if $tmp  is  zero:
    call AppendByte('S');  call AppendByte('Y');  call AppendByte('N');  call
     AppendByte('C');  call AppendByte('I');  call AppendByte('D');
    call AppendByte(' ');  call AppendByte(' ');
    goto GPLHaveOp;
  end
 $tmp = $kw - KW_CSWAP;
  if $tmp  is  zero:
    call AppendByte('C');  call AppendByte('S');  call AppendByte('W');  call
     AppendByte('A');  call AppendByte('P');
    call AppendByte(' ');  call AppendByte(' ');  call AppendByte(' ');
    goto GPLHaveOp;
  end
  call AppendByte('L');  call AppendByte('D');  call AppendByte('V');  call AppendByte('T');
   call AppendByte('S');
  call AppendByte(' ');  call AppendByte(' ');  call AppendByte(' ');
 GPLHaveOp:
 $tmp = $kw - KW_CSWAP;
  if $tmp  is  zero:
    call GenRegRef($x);
    goto GPLAddr;
  end
 $tmp = $kw - KW_LDVTS;
  if $tmp  is  zero:
    call GenRegRef($x);
    goto GPLAddr;
  end
  call GenConstExpr($x);
 GPLAddr:
  call AppendByte(',');
  call GenAddrYZ($base, $off);
  call AppendByte('\n');
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def GenSync($node):
  register $tmp, $x;
 $tmp = &StrIndent8;
  call AppendCStr($tmp);
  call AppendByte('S');  call AppendByte('Y');  call AppendByte('N');  call AppendByte('C');
  call AppendByte(' ');  call AppendByte(' ');  call AppendByte(' ');  call AppendByte(' ');
 $x =  call NodeGet($node, ND_A);
  call GenConstExpr($x);
  call AppendByte('\n');
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def GenTrip($node):
  register $tmp, $x, $y, $z;
 $tmp = &StrIndent8;
  call AppendCStr($tmp);
  call AppendByte('T');  call AppendByte('R');  call AppendByte('I');  call AppendByte('P');
  call AppendByte(' ');  call AppendByte(' ');  call AppendByte(' ');  call AppendByte(' ');
 $x =  call NodeGet($node, ND_A);
 $y =  call NodeGet($node, ND_B);
 $z =  call NodeGet($node, ND_C);
  call GenConstExpr($x);
  call AppendByte(',');
  call GenConstExpr($y);
  call AppendByte(',');
  call GenConstExpr($z);
  call AppendByte('\n');
  return;
end
```

⟨ ***Generator procedures*** ⟩ +≡

```
def GenResume($node):
  register $tmp, $z;
 $tmp = &StrIndent8;
  call AppendCStr($tmp);
  call AppendByte('R');  call AppendByte('E');  call AppendByte('S');  call AppendByte('U');
   call AppendByte('M');  call AppendByte('E');
  call AppendByte(' ');  call AppendByte(' ');
 $z =  call NodeGet($node, ND_A);
  if $z  is  zero:
    call AppendByte('0');
    call AppendByte('\n');
    return;
  end
  call GenConstExpr($z);
  call AppendByte('\n');
  return;
end
```

`GenStmtComment` writes “ `!  stmt-kind N` n” for a statement-kind stub. Distinct from `GenCommentKind` so that the indentation matches

the body-column of the surrounding procedure.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenStmtComment($kind):
  call AppendByte(' ');
  call AppendByte(' ');
  call AppendByte(' ');
  call AppendByte(' ');
  call AppendByte(' ');
  call AppendByte(' ');
  call AppendByte(' ');
  call AppendByte(' ');
  call AppendByte('!');
  call AppendByte(' ');
  call AppendByte('s');
  call AppendByte('t');
  call AppendByte('m');
  call AppendByte('t');
  call AppendByte('-');
  call AppendByte('k');
  call AppendByte('i');
  call AppendByte('n');
  call AppendByte('d');
  call AppendByte(' ');
  call AppendInt($kind);
  call AppendByte('\n');
  return;
end
```

Static strings used by the procedure-body emitters.

⟨ ***Data segment*** ⟩ +≡

```
byte StrDefOpen     "! def ", 0;
byte StrDefClose    "! end ", 0;
byte StrIsHere      "        IS      @\n", 0;
byte StrTrapDir     "        TRAP    ", 0;
byte StrStmtPrefix  "        ! stmt-kind ", 0;
```

`GenCommentKind` writes a line of the form "`!  node-kind N`
n" to `OutputBuf`. It is used by the generator's placeholder arms so that callers can see which AST kinds were visited during the walk.

⟨ ***Generator procedures*** ⟩ +≡

```
def GenCommentKind($kind):
  call AppendByte('!');
  call AppendByte(' ');
  call AppendByte('n');
  call AppendByte('o');
  call AppendByte('d');
  call AppendByte('e');
  call AppendByte('-');
  call AppendByte('k');
  call AppendByte('i');
  call AppendByte('n');
  call AppendByte('d');
```

```
  call AppendByte(' ');
  call AppendInt($kind);
  call AppendByte('\n');
  return;
end
```

Static strings used by the generator's placeholder emitters.

⟨ ***Data segment*** ⟩ +≡

```
byte GenKindPrefix  "! node-kind ", 0;
byte StrNodeLine    "! node-line ", 0;
byte StrPrefix      "        PREFIX  ", 0;
byte StrPool        "        POOL\n", 0;
byte StrLocal       "        LOCAL   ", 0;
byte StrSegData     "        LOC     Data_Segment\n", 0;
byte StrSegCode     "        LOC     #100\n", 0;
byte StrMainIs      "Main    IS      @\n", 0;
byte StrIndent8     "        ", 0;
byte StrLocDir      "        LOC     ", 0;
byte StrLocAtPlus   "        LOC     @+", 0;
byte StrIsDir       "        IS      ", 0;
byte StrGregDir     "        GREG    ", 0;
byte StrGregHere    "        GREG    @\n", 0;
byte StrByteDir     "        BYTE    ", 0;
byte StrWydeDir     "        WYDE    ", 0;
byte StrTetraDir    "        TETRA   ", 0;
byte StrOctaDir     "        OCTA    ", 0;
```

F MMIX and PL/MMIX Side-by-Side

"It seems to me that translating from one language into another, if it be not from the queens of languages, the Greek and Latin, is like looking at Flemish tapestries from the wrong side; for though the figures are visible, they are full of threads that make them indistinct, and they do not show with the smoothness and brightness of the right side; and translation from easy languages argues neither ingenuity nor command of words, any more than transcribing or copying out one document from another."

— Miguel de Cervantes, *Don Quixote*, Part II, Ch. 62 (1615); transl. John Ormsby (1885)

This appendix presents MMIX programs from Donald Knuth's *The Art of Computer Programming* alongside their PL/MMIX translations, shown side-by-side for comparison. The MMIX originals are drawn from the MMIXmasters collection; the PL/MMIX versions were written using the language described in this manual. Programs are grouped by the TAOCP section to which they belong.

TAOCP 1.3.2 – MMIX Assembly Language exercises

1-3-2–21: Farey series

`1-3-2-21.mms`

```
\def\date{05 Nov 2011}\def\source{V1F1, p.\ 47}\def\author{Udo We...
{ Farey series}
t         IS        $255
          LOC       Data_Segment
_x        GREG      @
          LOC       Data_Segment+4*10000
_y        GREG      @
          LOC       #100
       Computation of Farey series
       Calling sequence: SET \$1,$n$; PUSHJ \$0,:Farey
       Entry conditions: $n$ in \$1 is the order of the series, $1 < n <...
       l\_x and l\_y are arrays, each for at least 10000 tetras
       Exit conditions: \$0 is number of $x/y$ pairs, i.e., entries gene...
          PREFIX    :FAREY:
n         IS        $1                  Parameter: order of the Farey series
kk        IS        $2                  $kk \gets 4*k$
yk        IS        $3                  $y_k$
yk1       IS        $4                  $y_{k+1}$
xk        IS        $5                  $x_k$
xk1       IS        $6                  $x_{k+1}$
flr       IS        $7
yk2       IS        $8                  $y_{k+2}$
xk2       IS        $9                  $x_{k+2}$
:Farey    SET       n,$01! Get the parameter.
          SET       xk,01! Init for $k=0$.
          STTU      xk,:_x,4*01!
          SET       yk,11!
          STTU      yk,:_y,4*01!
          SET       xk1,11! Init for $k=1$.
          STTU      xk1,:_x,4*11!
          SET       yk1,n1!
          STTU      yk1,:_y,4*11!
          SET       kk,41! $k\gets1$.
nextval   ADDU      flr,yk,n            !A! Calculate the next values xk2 and yk2.
          DIVU      flr,flr,yk1         !A! $flr \gets \lfloor(yk+n)/yk1\rfloor$.
          MULU      xk2,flr,xk1A!
          SUBU      xk2,xk2,xkA! $\mmm xk2 \gets \mmm flr*\mmm xk1 - \mm xk$.
          MULU      yk2,flr,yk1A!
          SUBU      yk2,yk2,ykA! $\mmm yk2 \gets \mmm flr*\mmm yk1 - \mm yk$.
          INCL      kk,4A! $k \gets k + 1$.
          STTU      xk2,:_x,kkA!
          STTU      yk2,:_y,kkA!
          SET       xk,xk1A! Shuffle the registers.
          SET       xk1,xk2A!
          SET       yk,yk1A!
          SET       yk1,yk2A!
          CMPU      flr,xk2,yk2A! The computation stops when
          PBNZ      flr,nextval         !A!\bad1\bad $1 = 1/1$ is computed.
          INCL      kk,41!
          SR        kk,kk,21! Remove factor for tetra.
          SET       $0,kk1! The number of elements
          POP       1,0                 !1! is returned.
\endwAoA\bye
          PREFIX    :
       Output the Farey series to StdOut
       Calling sequence: SET \$1,$x$; PUSHJ \$0,:Output
       Entry condition: $x$ is the number of quotients to output, the da...
       Exit condition:
          PREFIX    :Output:
          LOC       (@+7)&-8
          GREG      @
Buf       BYTE      "    /   ",#a,0     Template for output
NoPairs   BYTE      "000 pairs x/y have been generated",#a,0
_template GREG      #202020202F202020   This is `` / ''
items     IS        $1                  Parameter: number of array entries
asc       IS        $2                  An ASCII char
idx       IS        $3                  Index to { l\_x} and { l\_y}
acc       IS        $4                  The accumulator
pos       IS        $5                  Position in Buf
:Output   SET       items,$0            Get parameter.
          SL        items,items,2       Work with tetras.
          SET       idx,0               Init index for data.
          SET       acc,$0              Print parameter.
          LDA       :t,NoPairs
          SET       pos,2
```

`1-3-2-21.plm`

```
# Farey series computation and output (TAOCP 1.3.2, Exercise 21)
# Farey(n): compute Farey series of order n (1 < n < 182)
#   stores x values in array $_x, y values in $_y
#   returns count of pairs
# Output(count): print the pairs
# PrintBanner(n): print "Farey series of order N\n"

# Based on an .mms file by Udo Wermuth
segment data;
greg $_x[40000];
greg $_y[40000];
# Output buffer: "    /   \n\0" template + 6 bytes padding = 16 b...
greg $BufBase;
byte BufData "    /   \n\0", 0, 0, 0, 0, 0, 0;
byte NoPairs "000 pairs x/y have been generated\n", 0;
# Banner buffer: "Farey series of order " + 3 digit slots + "\n\0...
greg $BannerBase;
byte BannerBase "Farey series of order ", 32, 32, 32, 10, 0;

# PrintBanner procedure: write n as decimal into banner buffer an...
# Parameter: $0 = n (order)
def PrintBanner($ord):
  register $t = $255;
  register $tmp, $digit, $slot;

  # Pre-fill the three digit slots (offsets 22, 23, 24) with spaces
  $BannerBase[22].bu = ' ';
  $BannerBase[23].bu = ' ';
  $BannerBase[24].bu = ' ';
  $slot = 24;   # start filling from the units position, right to left
  BannerDigit:
  $tmp = $ord / 10;
  $digit = $rR;
  $digit = $digit + '0';   # remainder → ASCII digit
  $BannerBase[$slot].bu = $digit;
  $slot = $slot - 1;
  $ord = $tmp;
  goto BannerDigit if likely $ord is nonzero;
  $t = $BannerBase;
  trap 0, FPUTS, STDOUT;
  return;
end

# Farey procedure: compute Farey series of order n, store in $_x,...
# Parameter: $0 = n; Returns: $0 = count of pairs
def Farey($ret):
  register $n;
  register $kk;# 4*k = tetra index
  register $yk;
  register $yk1;
  register $xk;
  register $xk1;
  register $flr;
  register $yk2;
  register $xk2;

  $n = $0;
  # Initialize k=0: (0/1)
  $xk = 0;
  [$_x].tu = $xk;   # X[0] = 0
  $yk = 1;
  [$_y].tu = $yk;   # Y[0] = 1
  # Initialize k=1: (1/n)
  $xk1 = 1;
  $_x[4].tu = $xk1;   # X[1] = 1
  $yk1 = $n;
  $_y[4].tu = $yk1;   # Y[1] = n
  $kk = 4;   # k = 1

  FareyLoop:
  $flr = $yk + $n;
  $flr = $flr /u $yk1;   # floor((yk + n) / yk1)
  $xk2 = $flr *u $xk1;
  $xk2 = $xk2 -u $xk;   # xk2 = flr*xk1 - xk
  $yk2 = $flr *u $yk1;
```

```
0H      DIV     acc,acc,10
        GET     asc,:rR
        INCL    asc,'0'             Get an ASCII digit
        STBU    asc,:t,pos          and store it into NoPairs.
        SUB     pos,pos,1
        PBNZ    acc,0B
        PBN     pos,2F              Fill leading zeros with blanks.
        SET     asc,' '
1H      STBU    asc,:t,pos
        SUB     pos,pos,1
        PBNN    pos,1B
2H      TRAP    0,:Fputs,:StdOut
7H      STOU    _template,Buf       Clear buffer.
        LDA     pos,Buf+2
        LDTU    acc,:_x,idx         Get value from array l\_x.
0H      DIV     acc,acc,10
        GET     asc,:rR
        INCL    asc,'0'             Convert to ASCII digit
        STBU    asc,pos,0           and store it into Buf.
        SUB     pos,pos,1
        PBNZ    acc,0B              Continue if $acc > 0$.
        LDA     pos,Buf+7           Now output the l\_y value.
        LDTU    acc,:_y,idx
0H      DIV     acc,acc,10
        GET     asc,:rR
        INCL    asc,'0'
        STBU    asc,pos,0
        SUB     pos,pos,1
        PBNZ    acc,0B
        LDA     :t,Buf
        TRAP    0,:Fputs,:StdOut    Output Buf and newline.
        INCL    idx,4               Next entries from the data arrays.
        CMP     :t,items,idx        Done?
        PBNZ    :t,7B               No, go back.
        POP     0,0                 Yes, return.
        PREFIX  :

ord     IS      $1                  parameter to Farey: order of the series
pairs   IS      ord                 parameter to Output: number of pairs
        GREG    @
Banner  BYTE    "Farey series of order "
Order   BYTE    "00",#a,0
Main    SET     ord,#2037           First set the order to 7.
        STWU    ord,:Order
        LDA     t,Banner
        TRAP    0,Fputs,StdOut      Print Banner.
        SET     ord,7
        PUSHJ   $0,:Farey           Call Farey.
        SET     pairs,$0            Get number of pairs
        PUSHJ   $0,:Output          and output the pairs.
        SET     ord,#3133           Next set the order to 13.
        STWU    ord,:Order
        LDA     t,Banner
        TRAP    0,Fputs,StdOut
        SET     ord,13
        PUSHJ   $0,:Farey
        SET     pairs,$0
        PUSHJ   $0,:Output
        SET     ord,#3339           Last test with order 39.
        STWU    ord,:Order
        LDA     t,Banner
        TRAP    0,Fputs,StdOut
        SET     ord,39
        PUSHJ   $0,:Farey
        SET     pairs,$0
        PUSHJ   $0,:Output
        TRAP    0,Halt,0
```

```
  $yk2 = $yk2 -u $yk;   # yk2 = flr*yk1 - yk
  $kk = $kk + 4;
  $_x[$kk].tu = $xk2;
  $_y[$kk].tu = $yk2;
  $xk = $xk1;
  $xk1 = $xk2;
  $yk = $yk1;
  $yk1 = $yk2;
  $flr = $xk2 cmpu $yk2;
  goto FareyLoop if likely $flr is nonzero;

  $kk = $kk + 4;
  $kk = $kk >>u 2;   # SR kk,kk,2: divide by 4 to get count
  $ret = $kk;
  return $ret;
end

# Output procedure: print 'items' pairs from $_x/$_y arrays
# Parameter: $0 = items (count)
def Output($0):
  register $items, $asc, $idx, $acc, $bufpos;
  register $t = $255;
  register $template, $tmp, $digit;

  $template = #202020202F202020;   # "    /   " - 8 bytes
  $items = $0;
  $items = $items <<u 2;   # work with tetras: items * 4
  $idx = 0;
  $acc = $0;   # print count first

  # Print number of pairs
  $t = &NoPairs;
  $bufpos = 2;   # 3-digit field: positions 2, 1, 0
  CountDigits:
  $tmp = $acc / 10;
  $asc = $rR;
  $asc = $asc + '0';   # ASCII digit
  $t[$bufpos].bu = $asc;   # store into NoPairs string
  $bufpos = $bufpos - 1;
  $acc = $tmp;
  goto CountDigits if likely $acc is nonzero;
  # Fill leading zeros with spaces
  goto PrintCount if likely $bufpos is neg;
  $asc = ' ';   # space
  FillSpaces:
  $t[$bufpos].bu = $asc;
  $bufpos = $bufpos - 1;
  goto FillSpaces if likely $bufpos is nonneg;
  PrintCount:
  trap 0, FPUTS, STDOUT;   # print NoPairs

  # Print each pair
  PairLoop:
  # Write template to Buf
  [$BufBase].o = $template;
  # Get x value, convert to decimal, store at Buf+2 (right-justifie...
  $bufpos = $BufBase + 2;
  $acc = $_x[$idx].tu;   # LDTU: load x[idx/4]
  XDigits:
  $tmp = $acc / 10;
  $asc = $rR;
  $asc = $asc + 48;
  [$bufpos].bu = $asc;
  $bufpos = $bufpos - 1;
  $acc = $tmp;
  goto XDigits if likely $acc is nonzero;
  # Get y value
  $bufpos = $BufBase + 7;
  $acc = $_y[$idx].tu;
  YDigits:
  $tmp = $acc / 10;
  $asc = $rR;
  $asc = $asc + 48;
  [$bufpos].bu = $asc;
  $bufpos = $bufpos - 1;
  $acc = $tmp;
  goto YDigits if likely $acc is nonzero;
  # Print BufData
  $t = $BufBase;
  trap 0, FPUTS, STDOUT;
  $idx = $idx + 4;
  $tmp = $items - $idx;
  goto PairLoop if likely $tmp is nonzero;
  return;
end

segment code;
register $t = $255;
register $ord = $1;
register $pairs = $1;# alias ord

$ord = 7;
call PrintBanner($ord);
$ord = 7;
$pairs = call Farey($ord);
$pairs = call Output($pairs);

$ord = 13;
call PrintBanner($ord);
$ord = 13;
$pairs = call Farey($ord);
$pairs = call Output($pairs);

$ord = 39;
call PrintBanner($ord);
$ord = 39;
$pairs = call Farey($ord);
$pairs = call Output($pairs);

trap 0, HALT, 0;
```

1-3-2–29: Josephus problem

1-3-2-29.mms

```
\def\date{12 Nov 2011}\def\source{V1F1, p.\ 48}\def\author{Udo We...
{ Josephus problem}
N         IS        24                  initial population $n < 256$
M         IS        11                  every $m$th person is executed; $m>1$
t         IS        $255
tt        GREG      0                   two additional temporary registers
ttt       GREG      0
sno       IS        $0                  Sequence number
cnt       IS        $1                  A counter
cur       IS        $2                  Current element
b         IS        $3                  Number of empty slots in out
out       IS        $4                  Output 8 characters
_mn       GREG      -N+1
_ones     GREG      #0101010101010101
          LOC       Data_Segment
_buf      GREG      @
          LOC       @+920               maximal $2*9+3*90+4*157=916$\thinspace bytes
_ctop     GREG      @+N-1               top of circle for m N bytes
          LOC       #100
Main      SET       cnt,_mn1! Store distance to next man in circle.
0H        STO       _ones,_ctop,cnt\lceil n/8\rceil! Link each byte with next byte.
          ADD       cnt,cnt,8\lceil n/8\rceil! 8 links done.
          PBNP      cnt,0B\lceil n/8\rceil!
          STB       _mn,_ctop,01! Last byte jumps back to the first.
          SET       sno,01! Start execution; no one excluded so far.
          SET       cur,_mn1! Start with first person.
1H        SET       cnt,M-2n-1! Count down $m-2$ links.
2H        LDB       tt,_ctop,cur        !P! Get distance from cur to next person
          ADD       cur,cur,tt          !P! and follow the link to this person.
          SUB       cnt,cnt,1P!
          PBNZ      cnt,2BP! n-1
          LDB       tt,_ctop,cur        !n-1! cur survives the round,
          ADD       cnt,cur,tt          !n-1! but the next man cnt is executed.
          LDB       t,_ctop,cntn-1! Get his neighbor.
          ADD       tt,tt,t             !n-1! Add it to previous distance for cur.
          STB       tt,_ctop,cur        !n-1! Store the new value for cur.
          ADD       cur,cnt,tn-1! Move to next man for next round.
          ADD       sno,sno,1n-1! Increase the sequence number
          STB       sno,_ctop,cnt       !n-1! and store it for the executed man.
          CMP       t,sno,N-1n-1! Is only one person left?
          PBN       t,1Bn-1! If not start count down again.
          SET       sno,N1! He is the last man
          STB       sno,_ctop,cur       !1! and gets the number $n$.
Output    SET       sno,_mn1! Start the output with first man.
          SET       b,8                 !1! 8 chars in out are empty.
          SET       cnt,0               !1! Offset in l\_buf to store out.
0H        LDB       t,_ctop,snon! Load next sequence number.
2H        DIV       t,t,10D! Extract
          GET       tt,rR               !D! the digits
          ADD       tt,tt,'0'           !D! from right
          SL        ttt,ttt,8           !D! to left
          OR        ttt,ttt,tt          !D! and store them in the register ttt.
4H        PBNZ      t,2BD! n
          SET       t,' 'n! Add a space in front of each sequence number.
          JMP       9Fn!
1H        AND       t,ttt,#ff           !D! Move ttt in reverse order to out.
          SR        ttt,ttt,8D!
9H        SL        out,out,8D+n!
          ADD       out,out,tD+n!
          SUB       b,b,1D+n!
          PBNZ      b,3FD+n! \lfloor(D+n)/8\rfloor
          STO       out,_buf,cnt        !\lfloor(D+n)/8\rfloor! Store the characters in sno\_buf
          ADD       cnt,cnt,8\lfloor(D+n)/8\rfloor!
          SRU       out,out,64\lfloor(D+n)/8\rfloor!
          SET       b,8\lfloor(D+n)/8\rfloor!
3H        PBNZ      ttt,1BD+n! n
          ADD       sno,sno,1n! Next sequence number.
          PBNP      sno,0Bn!
          SL        out,out,81! Add null byte to the final octabyte.
          STO       out,_buf,cnt1!
          LDA       t,_buf1! Output sequence numbers.
          TRAP      0,Fputs,StdOut1!
          TRAP      0,Halt,01!
\endwAoA\bye
```

1-3-2-29.plm

```
# Josephus problem (TAOCP 1.3.2, Exercise 29)
# N=24 people in a circle; every M=11th person is executed.
# Output: sequence numbers in order of position in original circl...
# Uses a byte-array circular linked list with signed byte offsets...

# Based on an .mms file by Udo Wermuth
segment data;
greg $_buf;   # base of circle + output buffer
reserve 960;   # 24 circle + 936 output; zero-initialised by simulator

segment code;
register $t = $255;
register $tt = $11;
register $ttt = $12;
register $sno = $0;# sequence number
register $cnt = $1;# counter
register $cur = $2;# current element (negative offset from ctop)
register $b = $3;# empty slots in out
register $out = $4;# output accumulator
register $ctop = $5;# top of circle = $_buf + N-1
register $ones = $8;# #0101010101010101
register $mn = $9;# -N+1 = -23
register $tmp = $10;

# Set up constants
$ones = #0101010101010101;
$mn = -23;   # -N+1 where N=24
$ctop = $_buf + 23;   # circle: $_buf[0..23], ctop = &$_buf[23]

# Initialize circle: each person links to next (offset = 1)
$cnt = $mn;   # cnt = -23
InitLoop:
$ctop[$cnt].o = $ones;   # STO: store 8 bytes of 0x01
$cnt = $cnt + 8;
goto InitLoop if $cnt is nonpos;
# Last person (at ctop+0) links back to first (at ctop-23): offse...
[$ctop].b = $mn;   # STB: store -23 as signed byte

# Main execution: N-1 times remove every M-th person
$sno = 0;
$cur = $mn;   # start at first person
ExecLoop:
$cnt = 9;   # count down M-2 = 11-2 = 9 links
CountDown:
$tt = $ctop[$cur].b;   # LDB: get link (distance to next)
$cur = $cur + $tt;   # follow link
$cnt = $cnt - 1;
goto CountDown if $cnt is nonzero;
# cur now points to person who survives; next person is executed
$tt = $ctop[$cur].b;   # cur's link
$cnt = $cur + $tt;   # cnt = index of executed person
$t = $ctop[$cnt].b;   # t = his neighbor's link
$tt = $tt + $t;   # combined distance: cur now skips executed person
$ctop[$cur].b = $tt;   # update cur's link to skip executed person
$cur = $cnt + $t;   # move to next person after executed
$sno = $sno + 1;   # sequence number of executed person
$ctop[$cnt].b = $sno;   # store his position in output order
$tmp = $sno - 23;   # N-1 = 23
goto ExecLoop if $tmp is neg;

# Last person remaining
$sno = 24;   # N = 24
$ctop[$cur].b = $sno;   # he is the last

# Output: traverse circle from first person, print sequence numbe...
$sno = $mn;   # start at first person (offset -23)
$b = 8;   # 8 chars fit in out
$cnt = 0;   # offset into $_buf for output storage

OutputLoop:
$t = $ctop[$sno].b;   # load sequence number of this person
# Convert t to decimal digits right-to-left, accumulate in ttt
$ttt = 0;
Digits:
$tmp = $t / 10;
$tt = $rR;
$tt = $tt + 48;   # ASCII digit
$ttt = $ttt <<u 8;
$ttt = $ttt | $tt;
$t = $tmp;
goto Digits if $t is nonzero;
# Add space before each sequence number
$t = 32;   # ' '
goto AccumChar;
OutputInner:
$t = $ttt & #ff;   # get next digit from ttt
$ttt = $ttt >>u 8;
AccumChar:
$out = $out <<u 8;
$out = $out + $t;
$b = $b - 1;
goto FlushCheck if $b is nonzero;
# store 8 chars
$_buf[$cnt].o = $out;
$cnt = $cnt + 8;
$out = $out >>u 64;   # SRU with 64 clears the register
$out = 0;
$b = 8;
FlushCheck:
goto OutputInner if $ttt is nonzero;
$sno = $sno + 1;   # next person
goto OutputLoop if $sno is nonpos;

# Flush remaining characters with null terminator
$out = $out <<u 8;   # add null byte
$_buf[$cnt].o = $out;

# Print output string
$t = $_buf;
$t = $t + 24;   # skip the 24-byte circle; output starts at $_buf+24
```

```
trap 0, FPUTS, STDOUT;
trap 0, HALT, 0;
```

TAOCP 1.3.3 – Applications to Permutations

1-3-3A: Program A – Permutation multiplication in cycle form

1-3-3A.mms

```
\def\date{03 Dec 2011}\def\source{V1, p.\ 168}\def\author{Udo Wer...
{ Program A} ({ Multiplication of permutations in cycle form})
t         IS        $255
tt        GREG      0
_lpren    GREG      #20202028          `` (''
_rpren    GREG      #20202029          `` )''
_tag      GREG      #80000000          Set MSB of a tetra
_untag    GREG      #7FFFFFFF          Turn off the MSB of a tetra
_nlnull   GREG      #0a000000          Newline and a zero byte
ip        IS        $2                 Pointer for input permutation
op        IS        $3                 Pointer for output permutation
p         IS        $4                 A symbol of the permutation
size      IS        $5                 The size of the input permutation
start     IS        $6                 The variables of the algorithm
current   IS        $7
          LOC       Data_Segment
          GREG      @
NoArg     BYTE      "Missing argument: file with input permutation expected",#a,0
NoFile    BYTE      "Can't open the file given in first argument.",#a,0
BUFSIZE   IS        80+1+1             80 Bytes plus newline can be read
INP       IS        3                  Handle for input file
ArgIn     OCTA      0,TextRead         First octabyte is later filled with argument
ArgRead   OCTA      0,BUFSIZE          Ditto
Perm      GREG      @                  Location to store the permutations
          LOC       #100
Error1    LDA       t,NoArg
          JMP       PrtAns
Error2    LDA       t,NoFile
          JMP       PrtAns
Main      SET       tt,Perm
          LDO       t,$1,8
          BZ        t,Error1           No argument: error
          STO       t,ArgIn            Otherwise use the argument.
OH        LDA       t,ArgIn            Open input file.
          TRAP      0,Fopen,INP
          BN        t,Error2           $-1$ indicates an error.
ReadLine  STO       tt,ArgRead         Read the input.
          LDA       t,ArgRead
          TRAP      0,Fgets,INP
          BN        t,EndRead
          ADD       tt,tt,t
          SUB       t,tt,t             Output the input line.
          TRAP      0,Fputs,StdOut
          SUB       tt,tt,1            Remove the newline byte.
          JMP       ReadLine
EndRead   TRAP      0,Fclose,INP       Close the input file.
          SUB       op,tt,Perm1! Start output after the equal sign.
          SUB       size,op,41!
          SET       ip,O1! $\mm ip\gets0$.
A1        LDT       p,Perm,ipA! \step A1. First pass. Load symbol into m p.
          CMP       t,p,_lprenA! Is m p a left parenthesis?
          PBNZ      t,OFA! B No, jump to test for right parenthesis.
          OR        p,p,_tagB! Yes, tag the left parenthesis.
          STTU      p,Perm,ipB!
          ADD       ip,ip,4B!
          LDT       p,Perm,ipB! Get the next symbol and
          OR        tt,p,_tag          !B! add a tag to it.
OH        CMP       t,p,_rprenC! Is it a right parenthesis?
          PBNZ      t,OFC! D No, test if the end is reached.
          STTU      tt,Perm,ipD! Yes, replace this parenthesis.
OH        ADD       ip,ip,4C!
          CMP       t,ip,sizeC!
          PBNZ      t,A1C! 1
          JMP       A21!
A6        STT       _rpren,Perm,opR! \step A6. Close. Output a right parenthesis.
          ADD       op,op,4R!
          SUB       tt,op,3*4R! Check for singleton cycle.
          LDT       p,Perm,ttR!
          CMP       t,p,_lprenR! Appears a `(' two tetras earlier?
          CSZ       op,t,ttR! Reset \mm op if yes.
A2        SET       ip,OE! \step A2. Open. Set \mm ip to the first element.
OH        ADD       ip,ip,4F! The leftist parenthesis is skipped.
          CMP       t,ip,sizeF!
          BZ        t,DoneF! 1 Exit at the end of input.
          LDT       p,Perm,ipG! Search untagged symbol.
          PBN       p,OBG! H Loop if tagged.
          SET       start,p            !H! Set start.
          STT       _lpren,Perm,opH! Output a left parenthesis.
          ADD       op,op,4H!
          STT       p,Perm,opH! Output the element
          ADD       op,op,4H!
          OR        p,p,_tag           !H! and tag it.
          STTU      p,Perm,ipH!
A3        ADD       ip,ip,4            !J! \step A3. Set CURRENT.
          LDT       p,Perm,ipJ! Get next element and
          AND       current,p,_untag   !J! store it without a tag in current.
          STT       current,Perm,sizeJ! Store it as sentinel.
A4        ADD       ip,ip,4K! \step A4. Scan formula.
          LDT       p,Perm,ipK! Load next symbol
          AND       p,p,_untag         !K! remove possible tag
          CMP       t,p,current        !K! and compare it to current.
          PBNZ      t,A4K! L
          CMP       t,ip,sizeL!
          BNN       t,A5L! P Branch if sentinel is reached.
          OR        p,p,_tag           !O! Element \m p equals current so tag it.
          STTU      p,Perm,ipO!
          JMP       A3O!
A5        CMP       t,current,start    !P! \steq A5. $CURRENT = START$?
          BZ        t,A6P! R Yes, close the output cycle.
          STT       current,Perm,op    !Q! No, output current.
          ADD       op,op,4Q!
          SET       ip,0Q! Start in A4 from the left.
          JMP       A4Q!
Done      ADD       size,size,4
          LDA       t,Perm,size        Start output after the equal sign.
          CMP       tt,op,size         Test if output is empty.
          BNZ       tt,1F
```

1-3-3A.plm

```
# Multiplication of permutations in cycle form (TAOCP 1.3.3, Algo...
# Reads a permutation formula from a file (first command-line arg...
# computes the product in cycle form, and prints the result.


# Based on an .mms file by Udo Wermuth
segment data;
greg $DataBase;
byte NoArg "Missing argument: file with input permutation expected\n", 0;
byte NoFile "Can't open the file given in first argument.\n", 0;
# $ArgIn: {filename_ptr, TextRead=5} for FOPEN
greg $ArgIn;
octa ArgInData 0, 5;
# $ArgRead: {buf_ptr, BUFSIZE=82} for FGETS
greg $ArgRead;
octa ArgReadData 0, 82;

greg $_lpren = #20202028;   # "   (" - left  parenthesis tetra
greg $_rpren = #20202029;   # "   )" - right parenthesis tetra
greg $_tag   = #80000000;   # MSB tag for tetras
greg $_untag = #7FFFFFFF;   # clear MSB of tetra
greg $_nlnull = #0a000000;   # newline + null byte

greg $Perm;   # permutation buffer starts here

segment code;
register $t = $255;
register $tt = $6;
register $ip, $op, $p, $size;
register $start = $6;# alias tt - only used in second pass
register $current, $tmp;

$tt = $Perm;
$t = $1[8].o;   # get first argument pointer from argv
goto Error1 if $t is zero;
[$ArgIn].o = $t;   # store filename pointer in $ArgIn struct
$t = $ArgIn;
trap 0, FOPEN, 3;   # INP=3
goto Error2 if $t is neg;

ReadLine:
[$ArgRead].o = $tt;   # set read buffer pointer
$t = $ArgRead;
trap 0, FGETS, 3;   # INP=3
goto EndRead if $t is neg;
$tt = $tt + $t;
$tmp = $tt - $t;
$t = $tmp;
trap 0, FPUTS, STDOUT;   # echo input line
$tt = $tt - 1;   # remove trailing newline
goto ReadLine;

EndRead:
$t = $DataBase;
trap 0, FCLOSE, 3;   # INP=3

$op = $tt - $Perm;   # byte offset of output start from $Perm
$size = $op - 4;   # byte offset of last symbol
$ip = 0;

# A1. First pass: tag parentheses
A1:
$p = $Perm[$ip].t;   # load symbol p from input
$tmp = $p - $_lpren;
goto A1check_rpren if $tmp is nonzero;
# p is left paren: tag it and next symbol
$p = $p | $_tag;
$Perm[$ip].tu = $p;
$ip = $ip + 4;
$p = $Perm[$ip].t;
$tt = $p | $_tag;
A1check_rpren:
$tmp = $p - $_rpren;
goto A1advance if $tmp is nonzero;
$Perm[$ip].tu = $tt;   # store tagged next-symbol as replacement
A1advance:
$ip = $ip + 4;
$tmp = $ip - $size;
goto A1 if $tmp is nonzero;
goto A2;

# A6. Close cycle: output right parenthesis
A6:
$Perm[$op].tu = $_rpren;
$op = $op + 4;
$tmp = $op - 12;   # check for singleton: 3 tetras back
$p = $Perm[$tmp].t;
$tmp = $p - $_lpren;
$op = $tmp if  $tmp is zero;   # CSZ: reset op if '(' two positions back

# A2. Open: find first untagged symbol
A2:
$ip = 0;
A2skip:
$ip = $ip + 4;
$tmp = $ip - $size;
goto Done if $tmp is zero;
$p = $Perm[$ip].t;
goto A2skip if $p is neg;
$start = $p;   # set START
$Perm[$op].tu = $_lpren;   # output '('
$op = $op + 4;
$Perm[$op].tu = $p;   # output element
$op = $op + 4;
$p = $p | $_tag;
$Perm[$ip].tu = $p;   # tag element
```

```
        STT     _lpren,t,0          Yes, so output the identity permutation.
        STT     _rpren,t,4
        ADD     op,size,8
1H      STT     _nlnull,Perm,op     Add newline and a null byte to output string.
PrtAns  TRAP    0,Fputs,StdOut
        TRAP    0,Halt,0
\endwAoA\bye
```

```
# A3. Set CURRENT
A3:
$ip = $ip + 4;
$p = $Perm[$ip].t;
$current = $p & $_untag;
$Perm[$size].tu = $current;   # store as sentinel

# A4. Scan formula for current
A4:
$ip = $ip + 4;
$p = $Perm[$ip].t;
$p = $p & $_untag;
$tmp = $p - $current;
goto A4 if $tmp is nonzero;
$tmp = $ip - $size;
goto A5 if $tmp is nonneg;
# found current in input: tag it and continue
$p = $p | $_tag;
$Perm[$ip].tu = $p;
goto A3;

# A5. Is CURRENT = START?
A5:
$tmp = $current - $start;
goto A6 if $tmp is zero;
$Perm[$op].tu = $current;   # output current
$op = $op + 4;
$ip = 0;
goto A4;

Done:
$size = $size + 4;
$tmp = $op - $size;
goto Done_nonempty if $tmp is nonzero;
# empty output: output "()"
$Perm[$size].tu = $_lpren;
$tmp = $size + 4;
$Perm[$tmp].tu = $_rpren;
$op = $size + 8;
Done_nonempty:
$Perm[$op].t = $_nlnull;   # append newline+null
$t = $Perm;
$t = $t + $size;   # point $255 to output string
goto PrtAns;

Error1:
$t = $DataBase;   # $255 = address of NoArg string
goto PrtAns;

Error2:
$tmp = $DataBase + 56;   # $255 = address of NoFile string (after 56 bytes of NoArg)
$t = $tmp;

PrtAns:
trap 0, FPUTS, STDOUT;
trap 0, HALT, 0;
```

1-3-3B: Program B – Permutation multiplication (table method)

1-3-3B.mms

```
\def\date{10 Dec 2011}\def\source{V1, p.\ 174}\def\author{Udo Wer...
{ Program B} ({ Multiplication of permutations in cycle form})
MAXWDS  IS      1200            Maximal size for table $T$
t       IS      $255
tt      GREG    0
_lpren  GREG    #20202028       `` ('' 
_rpren  GREG    #20202029       `` )''
_nlnull GREG    #0a000000       Newline and a zero byte
ip      IS      $2              Pointer for input permutation
op      IS      $3              Pointer for output permutation
p       IS      $4              A symbol of the permutation
size    IS      $5              The length of the input permutation
x       IS      $6              An element of the names table
z       IS      $7              The variables of the algorithm.
i       IS      $8
j       IS      $9
n       IS      $10             The number of different elements
        LOC     Data_Segment
        GREG    @
NoArg   BYTE    "Missing argument: file with input permutation expected",#a,0
NoFile  BYTE    "Can't open the file given in first argument.",#a,0
BUFSIZE IS      80+1+1          80 Bytes plus newline can be read
INP     IS      3               Handle for input file
ArgIn   OCTA    0,TextRead      First octabyte is later filled with argument
ArgRead OCTA    0,BUFSIZE       Ditto
X       GREG    @               Location to store the different elements
        LOC     @+MAXWDS
T       GREG    @               Location to store table $T$
        LOC     @+MAXWDS
Perm    GREG    @               Location to store the permutations
        LOC     #100
Error1  LDA     t,NoArg
        JMP     PrtAns
Error2  LDA     t,NoFile
        JMP     PrtAns
Main    SET     tt,Perm
        LDO     t,$1,8
        BZ      t,Error1        No argument: error
        STO     t,ArgIn         Otherwise use the argument.
OH      LDA     t,ArgIn         Open input file.
        TRAP    0,Fopen,INP
        BN      t,Error2        $-1$ indicates an error.
ReadLine STO    tt,ArgRead      Read the input.
        LDA     t,ArgRead
        TRAP    0,Fgets,INP
        BN      t,EndRead
        ADD     tt,tt,t
        SUB     t,tt,t          Output the input line.
        TRAP    0,Fputs,StdOut
        SUB     tt,tt,1         Remove the newline byte.
        JMP     ReadLine
EndRead TRAP    0,Fclose,INP    Close the input file.
        SUB     op,tt,Perm1! Start output after the equal sign.
        SET     size,op1! Remember the length of the permutation.
        SUB     ip,size,81! $\mm ip\gets\rb{last symbol of permutation}$.
        SET     n,41! $n\gets 1$, ready to store an element.
Right   SET     z,0A! Prepare for step B2, set $Z\gets0$.
B2      LDT     p,Perm,ipB! \step B2. Next element.
        SUB     ip,ip,4B!
        CMP     t,p,_prenB! Is it a right parenthesis?
        BZ      t,RightB! B-D
        CMP     t,p,_lprenD! Is it a left parenthesis?
        BZ      t,B4D! L
Search_p SET    i,nE! Prepare to search names table.
1H      SUB     i,i,4F!
        BNP     i,New_xF! H Branch if end of table is reached.
        LDT     x,X,iG! Get a known $x_i$.
        CMP     t,p,xG! Is it a match?
        PBNZ    t,1BG! J-H No, continue search.
B3      LDT     p,T,iJ! \step B3. Change $T[i]$.
        SET     t,zJ! Save the value of $Z$.
        STT     z,T,iJ! Exchange $Z$ with $T[i]$.
        SET     z,pJ!
        PBNZ    t,B2J! K Test former value of $Z$.
        SET     j,iK! $Z$ was zero, so $j\gets i$.
        JMP     B2K!
New_x   STT     p,X,nH! Insert a new element in names table.
        STT     n,T,nH! Initialize $T[n]$.
        SET     i,nH! $i\gets n$.
        ADD     n,n,4H! $n\gets n+1$.
        JMP     B3H! Now continue with step B3.
B4      STT     z,T,jL! \step B4. Change $T[j]$.
        PBP     ip,B2L! 1
Output  SET     i,n1! Output the permutation in cycle form.
OH      STT     _lpren,Perm,opP! Start with a cycle.
        ADD     op,op,4P!
1H      SUB     i,i,4Q!
        BNP     i,DoneQ! 1 Branch if table of names is processed.
        LDT     x,X,iT! Get an element name.
        BN      x,1BT! Q-P Branch if it already tagged.
2H      STT     x,Perm,opR! Otherwise output the element.
        ADD     op,op,4R!
        NEG     x,0,xR! Tag element
        STT     x,X,i           !R! and store it.
        LDT     i,T,iR! Load the successor.
        LDT     x,X,iR! Load the name of that element.
        PBP     x,2BR! S Branch if name is not tagged.
        STT     _rpren,Perm,opS! Otherwise close cycle.
        ADD     op,op,4S!
        SUB     tt,op,3*4S! Check for singleton cycle.
        LDT     p,Perm,ttS!
        CMP     t,p,_lprenS! Appears a `(' two tetras earlier?
        CSZ     op,t,ttS! Reset \mm op if yes.
        JMP     0BS!
Done    LDA     t,Perm,size     Start output after the equal sign.
        SUB     op,op,4
        CMP     tt,op,size      Test if output is empty.
        BNZ     tt,1F
        STT     _lpren,t,0      Yes, so output the identity permutation.
        STT     _rpren,t,4
```

1-3-3B.plm

```
# Multiplication of permutations in cycle form (TAOCP 1.3.3, Algo...
# Reads a permutation formula from a file (first command-line arg...
# uses table $T of successors, and prints the cycle-form product.
#
# MAXWDS=1200 tetras for tables $X and $T

# Based on an .mms file by Udo Wermuth
segment data $DataBase;
byte NoArg "Missing argument: file with input permutation expected\n", 0;
byte NoFile "Can't open the file given in first argument.\n", 0;
greg $ArgIn;
octa ArgInData 0, 5;   # {filename_ptr, TextRead=5} for FOPEN
greg $ArgRead;
octa ArgReadData 0, 82;   # {buf_ptr, BUFSIZE=82} for FGETS

greg $_lpren  = #20202028;   # "   ("
greg $_rpren  = #20202029;   # "   )"
greg $_nlnull = #0a000000;   # newline + null byte

greg $X;   # table of element names - MAXWDS=1200 tetras
reserve 4800;    # 1200 tetras; zero-initialised by simulator
greg $T;   # successor table - MAXWDS=1200 tetras
reserve 4800;    # 1200 tetras; zero-initialised by simulator
greg $Perm;   # permutation buffer

segment code;
register $t = $255;
register $tt = $6;
register $ip, $op, $p, $size;
register $x = $6;# alias tt - names table element
register $z, $i, $j;
register $n = $10;# number of distinct elements * 4
register $tmp = $11;

$tt = $Perm;
$t = $1[8].o;   # get first argument pointer from argv
goto Error1 if $t is zero;
[$ArgIn].o = $t;
$t = $ArgIn;
trap 0, FOPEN, 3;   # INP=3
goto Error2 if $t is neg;

ReadLine:
[$ArgRead].o = $tt;
$t = $ArgRead;
trap 0, FGETS, 3;   # INP=3
goto EndRead if $t is neg;
$tt = $tt + $t;
$tmp = $tt - $t;
$t = $tmp;
trap 0, FPUTS, STDOUT;
$tt = $tt - 1;
goto ReadLine;

EndRead:
$t = $DataBase;
trap 0, FCLOSE, 3;   # INP=3

$op = $tt - $Perm;   # B1. op = byte length of input
$size = $op;   # remember total length
$ip = $size - 8;   # ip <- last symbol offset
$n = 4;   # n <- 1 (ready for first element), * 4

# B2. Process right-to-left
Right:
$z = 0;   # Z <- 0
B2:
$p = $Perm[$ip].t;   # load symbol
$ip = $ip - 4;
$tmp = $p - $_rpren;
goto Right if $tmp is zero;
$tmp = $p - $_lpren;
goto B4 if $tmp is zero;

# Search for p in names table
Search_p:
$i = $n;
B2search:
$i = $i - 4;
goto New_x if $i is nonpos;
$x = $X[$i].t;   # load known element name
$tmp = $p - $x;
goto B2search if $tmp is nonzero;

# B3. Change $T[i]
B3:
$p = $T[$i].t;   # p <- $T[i]
$tmp = $z;   # save Z
$T[$i].tu = $z;   # $T[i] <- Z
$z = $p;   # Z <- old $T[i]
goto B2 if $tmp is nonzero;
$j = $i;   # Z was zero; j <- i
goto B2;

# New element: insert in names table
New_x:
$X[$n].tu = $p;   # $X[n] <- p
$T[$n].tu = $n;   # $T[n] <- n (self-loop)
$i = $n;
$n = $n + 4;
goto B3;

# B4. Change $T[j]
B4:
$T[$j].tu = $z;
goto B2 if $ip is pos;

# Output the permutation in cycle form
Output:
$i = $n;
```

```
          ADD       op,size,8
1H        STT       _nlnull,Perm,op     Add newline and a null byte to output string.
PrtAns    TRAP      0,Fputs,StdOut
          TRAP      0,Halt,0
\endwAoA\bye
```

```
OuterCycle:
$Perm[$op].tu = $_lpren;   # start cycle
$op = $op + 4;
InnerCycle:
$i = $i - 4;
goto Done if $i is nonpos;
$x = $X[$i].t;   # load element name
goto InnerCycle if $x is neg;   # already tagged, skip
OutElement:
$Perm[$op].tu = $x;   # output element
$op = $op + 4;
$x = - $x;   # negate = tag element
$X[$i].tu = $x;
$i = $T[$i].t;   # i <- successor index
$x = $X[$i].t;   # load successor name
goto OutElement if $x is pos;
# close cycle
$Perm[$op].tu = $_rpren;
$op = $op + 4;
$tmp = $op - 12;   # check singleton: 3 tetras back
$p = $Perm[$tmp].t;
$tmp = $p - $_lpren;
$op = $tmp if $tmp is zero;   # CSZ: remove singleton cycle
goto OuterCycle;

Done:
$tmp = $op - 4;
$size = $size;
$tmp = $op - $size;
goto Done_nonempty if $tmp is nonzero;
# empty output: identity "()"
$Perm[$size].tu = $_lpren;
$tmp = $size + 4;
$Perm[$tmp].tu = $_rpren;
$op = $size + 8;
Done_nonempty:
$Perm[$op].tu = $_nlnull;   # append newline+null
$t = $Perm;
$t = $t + $size;
goto PrtAns;

Error1:
$t = $DataBase;
goto PrtAns;

Error2:
$tmp = $DataBase + 56;
$t = $tmp;

PrtAns:
trap 0, FPUTS, STDOUT;
trap 0, HALT, 0;
```

1-3-3I: Program I – Invert permutation in place

1-3-3I.mms

```
\def\date{04 Jan 2012}\def\source{V1, p.\ 177}\def\author{Udo Wer...
{ Program I} ({ Inverse in place})
N          IS        6                   Number of elements in the permutation
j          IS        $2                  Variables of the algorithm
i          IS        $3
mm         IS        $4                  For $m$ the value is multiplied by 8
           LOC       Data_Segment
X          GREG      @
           OCTA      0                   $X[0]$ is not used
           OCTA      6,2,1,5,4,3         The data of Table 1.3.3--3
           LOC       #100
* Inverse a permutation in place
Invert     SET       mm,N1! \step I1. Initialize.
           SL        mm,mm,31! $m\gets n$.
           NEG       j,11! $j\gets-1$.
2H         LDO       i,X,mmN! \step I2. Next element. $i\gets X[m]$.
           PBN       i,5FN! C To I5 if $i<0$.
3H         STO       j,X,mmN! \step I3. Invert one. $X[m]\gets j$.
           SR        j,mm,3N!
           NEG       j,jN! $j\gets-m$.
           SL        mm,i,3N! $m\gets i$.
           LDO       i,X,mmN! $i\gets X[m]$.
4H         PBP       i,3BN! C \steq I4. End of cycle? To I3 if $i>0$.
           SET       i,jC! Otherwise set $i\gets j$.
5H         NEG       i,iN! \step I5. Store final value.
           STO       i,X,mmN! $X[m]\gets-i$.
6H         SUB       mm,mm,8N! \step I6. Loop on $m$.
           PBP       mm,2BN! 1 To I2 if $m>0$. \stopnumbering
* inspect memory locations of array X for the result
           TRAP      0,Halt,0
Main       IS        Invert
\endwAoA\bye
```

1-3-3I.plm

```
# Inverse a permutation in place (TAOCP 1.3.3, Algorithm I)
# N = 6 elements; $X[0] unused, $X[1..N] hold the permutation.

# Based on an .mms file by Udo Wermuth
segment data $X;
octa X 0, 6, 2, 1, 5, 4, 3;   # $X[0] unused; permutation in $X[1..6]

segment code;
register $mm = $4;
register $j, $i;
register $t = $5;

$mm = 6;
$mm = $mm << 3;   # I1. Initialize. m <- n
$j = -1;

I2:
$i = $X[$mm].o;   # I2. Next element. i <- $X[m]
goto I5 if $i is neg;
I3:
$X[$mm].o = $j;   # I3. Invert one. $X[m] <- j
$j = $mm >> 3;
$j = - $j;   # j <- -m
$mm = $i << 3;   # m <- i
$i = $X[$mm].o;   # i <- $X[m]
I4:
goto I3 if $i is pos;
$i = $j;   # I4. End of cycle? if i<=0 set i<-j
I5:
$i = - $i;   # I5. Store final value. $X[m] <- -i
$X[$mm].o = $i;
I6:
$mm = $mm - 8;   # I6. Loop on m.
goto I2 if $mm is pos;
trap 0, HALT, 0;
```

1-3-3J: Program J – Invert permutation in place (tag-free)

1-3-3J.mms

```
\def\date{06 Jan 2012}\def\source{V1, p.\ 178}\def\author{Udo Wer...
{ Program J} ({ Inverse in place})
N         IS        6                 Number of elements in the permutation
t         IS        $255              Temporary storage
i         IS        $2                Variables of the algorithm
ii        IS        i                 Two characters indicate that the
jj        IS        $3                variable is multiplied by 8
k         IS        jj
mm        IS        $4
          LOC       Data_Segment
X         GREG      @
          OCTA      0                 $X[0]$ is not used
          OCTA      6,2,1,5,4,3       The data of Table 1.3.3--4
          LOC       #100
* Inverse a permutation in place
Invert    SET       t,N1! \step J1. Negate all. $t\gets n$.
          SL        mm,t,31! $m\gets n$.
1H        LDO       k,X,mmN!
          NEG       k,kN!
          STO       k,X,mmN! $X[k]\gets -X[k]$.
          SUB       mm,mm,8N! $m\gets m-1$.
          PBP       mm,1BN! 1 Branch if $m>0$.
          SL        mm,t,31! $m\gets n$.
2H        SR        i,mm,3N! \step J2. Initialize. $i\gets m$.
3H        SL        jj,i,3A! \step J3. Find neagtive entry. $j\gets i$.
          LDO       i,X,jjA! $i\gets X[j]$.
          PBP       i,3BA! N To J3 if $i>0$.
4H        NEG       i,iN! \step J4. Invert.
          SL        ii,i,3N!
          LDO       t,X,iiN! $t\gets X[-i]$
          STO       t,X,jj            !N! and $X[j]\gets X[-i]$.
          SR        t,mm,3N! $t\gets m$
          STO       t,X,ii            !N! and $X[-i]\gets m$.
5H        SUB       mm,mm,8N! \step J5. Loop on $m$. $m\gets m-1$.
          PBP       mm,2BN! 1 Branch if $m>0$.\stopnumbering
* inspect memory locations of array X for the result
          TRAP      0,Halt,0
Main      IS        Invert
\endwAoA\bye
```

1-3-3J.plm

```
# Inverse a permutation in place (TAOCP 1.3.3, Algorithm J)
# N = 6 elements; $X[0] unused, $X[1..N] hold the permutation.

# Based on an .mms file by Udo Wermuth
segment data $X;
octa X 0, 6, 2, 1, 5, 4, 3;   # $X[0] unused; permutation in $X[1..6]

segment code;
register $t = $29;
register $i, $jj;
register $k = $3;# alias jj
register $mm, $tmp;

$t = 6;
$mm = $t << 3;   # J1. Negate all. m <- n
do:
  $k = $X[$mm].o;
  $k = - $k;
  $X[$mm].o = $k;   # $X[m] <- -$X[m]
  $mm = $mm - 8;
end while likely $mm is pos;

$mm = $t << 3;   # reset m <- n
do:   # J2/J5. Loop on m.
  $i = $mm >> 3;   # J2. Initialize. i <- m
  while likely $i is pos:   # J3. Find negative entry.
    $jj = $i << 3;
    $i = $X[$jj].o;   # i <- $X[j]
  end
  # J4. Invert.
  $i = - $i;
  $tmp = $i << 3;
  $t = $X[$tmp].o;   # t <- $X[-i]
  $X[$jj].o = $t;   # $X[j] <- $X[-i]
  $t = $mm >> 3;   # t <- m
  $X[$tmp].o = $t;   # $X[-i] <- m
  $mm = $mm - 8;   # J5. m <- m-1
end while likely $mm is pos;
trap 0, HALT, 0;
```

TAOCP 2.3.2 – Binary Tree Traversal

S232PD-DIFF: Program D – Symbolic polynomial differentiation

S232PD-DIFF.mms

```
% Section 2.3.2, Program D : Differentiation Routines
% Author Chan Vinh VONG <chanvinh.vong@gmail.com>
% Draft Version 2012 April 14th
% Refined       2012 April 19th
% Refined       2012 May   08th
% Refined       2012 July  08th
%
%                 >|<-----OCTA----->|<
%                  +---------------+
%                  | RLINK/63 RTAG/1| BA+0
%                  +---------------+
%                  | LLINK/63 LTAG/1| BA+8
%                  +---------------+
%                  | INFO/60 TYPE/4 | BA+16
%                  +---------------+
%
% Remarks:
% - INFO and TYPE are stored in the same octa, resp. w/ 60 and 4 bits
% - one data block is made of 3 octas
%

% Global Variables:
%     :Y tree head

        PREFIX DIFF:

dRLINK      IS 8*0
dLLINK      IS 8*1
dINFOTYPE   IS 8*2

P       IS $0
P1      IS $1
P2      IS $2
Q1      IS $3
Q       IS $4
info    IS $5
type    IS $6
warp    IS $7
ret     IS $8
t       IS $9

:DIFF   LDOU  P1,:Y,dLLINK  % D1. Initialize
1H      SET   P,P1
        LDOU  P1,P,dLLINK
        BNZ   P1,1B          % Y$ is the leftmost node

        % Coroutine Test
        % GO :a,:b,0

D2      LDOU  P1,P,dLLINK
        BZ    P1,2F
        LDOU  Q1,P1,dRLINK
2H      LDOU  type,P,dINFOTYPE
        AND   type,type,#F
        INCL  type,1
switch  GREG  @
        STBU  type,switch,7
        JMP   @+4
        JMP   CONST
        JMP   VAR
        JMP   LN
        JMP   NEG
        JMP   ADD
        JMP   MINUS
        JMP   TIMES
        JMP   DIV
        JMP   PWR

D3      CMP   t,type,4
        BNP   t,D4
        STOU  P2,P1,dRLINK

D4      SET   P2,P
        LDOU  P,P,dRLINK
        BOD   P,4F          % it was a right thread hence we got P$
Scan    LDOU  t,P,dLLINK    % else hard link thus get leftmost node
        BZ    t,4F
        SET   P,t
        JMP   Scan
4H      LDOU  t,P2,dRLINK
        BOD   t,D5
        STOU  Q,P2,dRLINK

D5      SRU   t,t,1
        SLU   t,t,1         % reset thread flag
        CMP   t,t,:Y
        BNZ   t,D2
        STOU  Q,:DY,dLLINK
        XOR   t,:DY,#1
        STOU  t,Q,dRLINK

        POP 0,0

XValue  IS #58

N_CONST0 GREG #0       % 0
N_CONST1 GREG #10      % 1
N_CONST2 GREG #20      % 2
N_VARX   GREG #581     % "X"
N_VARY   GREG #591     % "Y"
N_LN     GREG #4C4E2   % "LN"
N_NEG    GREG #4E45473 % "NEG"
N_PLUS   GREG #2B4     % "+"
N_MINUS  GREG #2D5     % "-"
N_TIMES  GREG #2A6     % "*"
N_DIV    GREG #2F7     % "/"
N_PWR    GREG #5E8     % "^"
```

S232PD-DIFF.plm

```
# Symbolic differentiation (TAOCP 2.3.2, Program D)
# DIFF procedure: traverses an expression tree and builds the der...
# Tree node layout (3 octabytes per node at BA):
#   BA+0:  RLINK (63 bits) | RTAG (1 bit)   - right child pointer...
#   BA+8:  LLINK (63 bits) | LTAG (1 bit)   - left child pointer ...
#   BA+16: INFO  (60 bits) | TYPE (4 bits)   - value and type cod...
#
# TYPE codes: 0=CONST, 1=VAR, 2=LN, 3=NEG, 4=ADD, 5=MINUS, 6=TIME...
# INFO: for VAR type, #58 = 'X'; for CONST, the numeric value
#
# Subroutines required by the original MMIX routine:
#   TREE0(info) - allocate leaf node with given INFO|TYPE
#   TREE1(type, child) - allocate unary node
#   TREE2(type, left, right) - allocate binary node
#   COPYRTBT(node) - copy subtree rooted at node (right-thread ba...
#   MULT(left, right) - build TIMES(left, right) tree
#   AGC_GC(node) - garbage collect tree rooted at node
#
# The original MMIX source expects these helpers to come from sur...
# This PL/MMIX version provides concrete implementations so the e...
# buildable with the local toolchain, which does not support exte...
#
# Global: $Y = tree head; $DY = derivative tree head

# Based on an .mms file by Chan Vinh VONG
greg $Y;   # tree head: input expression
greg $DY;   # derivative tree head: output
greg $NodePool;   # pool of tree node storage
greg $NodeNext = 0;   # next unused node
greg $NodeFree = 0;   # free-list head for reclaimed nodes
greg $DiffSaveP = 0;   # preserve current source node across scalar helper calls
greg $DiffSaveRJ = 0;   # preserve return address across scalar helper calls
reserve 4096;

# Field offsets within a node (octa-aligned)
register $dRLINK = $20;# = 0  - used as immediate 0
register $dLLINK = $21;# = 8  - used as immediate 8
register $dINFOTYPE = $22;# = 16 - used as immediate 16

# INFO/TYPE constants for tree node construction
register $N_CONST0 = $23;# 0 - constant 0
register $N_CONST1 = $24;# #10 - constant 1 (INFO=1, TYPE=0)
register $N_CONST2 = $25;# #20 - constant 2
register $N_VARX = $26;# #581 - variable X
register $N_LN = $27;# #4C4E2 - LN node info
register $N_NEG = $28;# #4E45473 - NEG node info
register $N_PLUS = $29;# #2B4 - + node info
register $N_MINUS = $30;# #2D5 - - node info
register $N_TIMES = $31;# #2A6 - * node info
register $N_DIV = $32;# #2F7 - / node info
register $N_PWR = $33;# #5E8 - ^ node info

# Main DIFF procedure
def DIFF():
  register $P;
  register $P1;
  register $P2;
  register $Q1;
  register $Q;
  register $info;
  register $typev;
  register $warp;
  register $ret;
  register $t;
  register $tmp;
  register $XValue = $14;# #58 = 'X'

  # Set up constant
  $XValue = #58;

  # Initialize node-type lookup constants
  $N_CONST0 = 0;
  $N_CONST1 = #10;
  $N_CONST2 = #20;
  $N_VARX   = #581;
  $N_LN     = #4C4E2;
  $N_NEG    = #4E45473;
  $N_PLUS   = #2B4;
  $N_MINUS  = #2D5;
  $N_TIMES  = #2A6;
  $N_DIV    = #2F7;
  $N_PWR    = #5E8;

  # D1. Initialize: find leftmost node of $Y
  $P1 = $Y[8].o;   # P1 = LLINK($Y)
  D1loop:
  $P = $P1;
  $P1 = $P[8].o;   # P1 = LLINK(P)
  goto D1loop if $P1 is nonzero;

  # Main traversal loop
  D2:
  $P1 = $P[8].o;   # P1 = LLINK(P)
  goto D2_noleft if $P1 is zero;
  $Q1 = [$P1].o;   # Q1 = RLINK(P1)
  D2_noleft:

  # Get type: INFO/TYPE octa, extract low 4 bits
  $typev = $P[16].o;
  $typev = $typev & #F;
  $typev = $typev + 1;   # type+1 for 1-based jump table

  # Switch on type (1-based: 1=CONST, 2=VAR, 3=LN, 4=NEG, 5=ADD, 6=...
  $tmp = $typev - 1;
  goto D_CONST if $tmp is zero;
  $tmp = $typev - 2;
  goto D_VAR if $tmp is zero;
```

```
%== NULLARY OPERATORS ================================================

CONST   JMP 1F
VAR     LDOU  info,P,dINFOTYPE
        SRU   info,info,4
        CMP   t,info,XValue
        BNZ   t,1F
        SET   t+2,N_CONST1
        JMP   2F
1H      SET   t+2,N_CONST0
2H      GET   warp,:rJ
        PUSHJ t+1,:TREE0
        PUT   :rJ,warp
        SET   Q,t+1
        JMP   D3

%== UNARY OPERATORS ==================================================

LN      LDOU  info,Q,dINFOTYPE
        BZ    info,D3
        GET   warp,:rJ
        SET   t+2,P1
        PUSHJ t+1,:COPYRTBT
        SET   t+4,t+1
        SET   t+3,Q
        SET   t+2,N_DIV
        PUSHJ t+1,:TREE2
        PUT   :rJ,warp
        SET   Q,t+1
        JMP D3

NEG     LDOU  info,Q,dINFOTYPE
        BZ    info,3F   % NEG 0 = 0
        SET   t+3,Q
        SET   t+2,N_NEG
        GET   warp,:rJ
        PUSHJ t+1,:TREE1
        PUT   :rJ,warp
        SET   Q,t+1
3H      JMP   D3

%== BINARY OPERATORS =================================================

ADD     LDOU  info,Q1,dINFOTYPE
        BNZ   info,4F
        SET   t,Q1              % dU = 0 => dU to be GC'd
        JMP   GC4
4H      LDOU  info,Q,dINFOTYPE
        BNZ   info,4F
        SET   t,Q               % dU != 0 and dV = 0 => dV to be GC'd
        SET   Q,Q1
GC4     GET   warp,:rJ          % garbage collect unused block
        SET   t+1,t
        PUSHJ t,:AGC:GC
        PUT   :rJ,warp
        JMP   D3
4H      GET   warp,:rJ          % dU != 0 and dV != 0
        SET   t+4,Q
        SET   t+3,Q1
        SET   t+2,N_PLUS
        PUSHJ t+1,:TREE2
        SET   Q,t+1
        PUT   :rJ,warp
        JMP   D3

MINUS   LDOU  info,Q,dINFOTYPE
        BNZ   info,5F
        SET   t,Q               % dV = 0 => GC dV
        SET   Q,Q1
        GET   warp,:rJ          % garbage collect unused block
        SET   t+1,t
        PUSHJ t,:AGC:GC
        PUT   :rJ,warp
        JMP   D3
5H      LDOU  info,Q1,dINFOTYPE
        BNZ   info,5F
        SET   t,Q1              % dV != 0 but dU = 0
        GET   warp,:rJ          % garbage collect unused block
        SET   t+1,t
        PUSHJ t,:AGC:GC
        PUT   :rJ,warp
        SET   t+3,Q
        SET   t+2,N_NEG
        PUSHJ t+1,:TREE1
        SET   Q,t+1
        PUT   :rJ,warp
        JMP   D3
5H      GET   warp,:rJ          % dV != 0 and dU != 0
        SET   t+4,Q
        SET   t+3,Q1
        SET   t+2,N_MINUS
        PUSHJ t+1,:TREE2
        SET   Q,t+1
        PUT   :rJ,warp
        JMP   D3

TIMES   LDOU  info,Q1,dINFOTYPE
        BZ    info,6F
        GET   warp,:rJ           % dU != 0
        SET   t+1,P2             % copy V
        PUSHJ t,:COPYRTBT
        SET   t+2,t              % MULT(dU,copy(V))
        SET   t+1,Q1
        PUSHJ t,:MULT
        SET   Q1,t
        PUT   :rJ,warp
6H      LDOU  info,Q,dINFOTYPE
        BZ    info,6F
        GET   warp,:rJ           % dV != 0
        LDOU  t,P1,dRLINK        % temp shunt rlink(P1) because
        STCO  1,P1,dRLINK        % we are still in the middle of the
                                 %   P1-Q1 temp bridging
        SET   t+2,P1             % copy U
        PUSHJ t+1,:COPYRTBT
        STOU  t,P1,dRLINK        % restore bridge
```

```
$tmp = $typev - 3;
goto D_LN if $tmp is zero;
$tmp = $typev - 4;
goto D_NEG if $tmp is zero;
$tmp = $typev - 5;
goto D_ADD if $tmp is zero;
$tmp = $typev - 6;
goto D_MINUS if $tmp is zero;
$tmp = $typev - 7;
goto D_TIMES if $tmp is zero;
$tmp = $typev - 8;
goto D_DIV if $tmp is zero;
# PWR
goto D_PWR;

D3:
# D3. If type > 4, link P2 into tree
$tmp = $typev - 4;
goto D4 if $tmp is nonpos;
[$P1].o = $P2;   # RLINK(P1) = P2

D4:
# D4. Advance P
$P2 = $P;
$P = [$P].o;   # P = RLINK(P) - follow right link or thread
$tmp = $P & 1;   # check if it's a thread (odd address)
goto D4_thread if $tmp is odd;
# Hard link: scan left to leftmost node
D4_scan:
$t = $P[8].o;   # LLINK(P)
goto D4_end if $t is zero;
$P = $t;
goto D4_scan;
D4_thread:
# It was a right thread
D4_end:

$t = [$P2].o;   # RLINK(P2)
$tmp = $t & 1;   # is it a thread?
goto D5 if $tmp is odd;
[$P2].o = $Q;   # STOU Q, P2, dRLINK

D5:
$t = $t >>u 1;
$t = $t <<u 1;   # clear thread bit
$tmp = $t - $Y;   # compare to $Y (tree head)
goto D2 if $tmp is nonzero;
# Done: link Q into $DY
$DY[8].o = $Q;   # STOU Q, $DY, dLLINK
$t = $DY ^ 1;
[$Q].o = $t;   # STOU t, Q, dRLINK - set thread back to $DY
return;

# CONST: d/dx(c) = 0
D_CONST:
$DiffSaveP = $P;
$DiffSaveRJ = $rJ;
$Q = call TREE0($N_CONST0);
$rJ = $DiffSaveRJ;
$P = $DiffSaveP;
goto D3;

# VAR: d/dx(X) = 1, d/dx($Y) = 0
D_VAR:
$info = $P[16].o;
$info = $info >>u 4;   # get INFO bits
$tmp = $info - $XValue;
goto D_VAR_zero if $tmp is nonzero;
$DiffSaveP = $P;
$DiffSaveRJ = $rJ;
$Q = call TREE0($N_CONST1);   # derivative of X is 1
$rJ = $DiffSaveRJ;
$P = $DiffSaveP;
goto D3;
D_VAR_zero:
$DiffSaveP = $P;
$DiffSaveRJ = $rJ;
$Q = call TREE0($N_CONST0);   # derivative of other var is 0
$rJ = $DiffSaveRJ;
$P = $DiffSaveP;
goto D3;

# LN: d/dx(ln u) = u'/u
D_LN:
$info = $Q[16].o;
goto D3 if $info is zero;   # Q = 0: d(ln u) = 0/u = 0
$warp = $rJ;
$t = call COPYRTBT($P1);   # copy subtree rooted at P1
$Q = call TREE2($N_DIV, $Q, $t);
$rJ = $warp;
goto D3;

# NEG: d/dx(-u) = -(u')
D_NEG:
$info = $Q[16].o;
goto D3 if $info is zero;
$warp = $rJ;
$tmp = $Q;
$Q = call TREE1($N_NEG, $tmp);
$rJ = $warp;
goto D3;

# ADD: d/dx(u+v) = u' + v' (simplify if either is 0)
D_ADD:
$info = $Q1[16].o;
goto D_ADD_both if $info is nonzero;
# Q1 = 0: result is Q
$t = $Q1;
$warp = $rJ;
$t = call AGC_GC($t);   # garbage collect Q1
$rJ = $warp;
goto D3;
D_ADD_both:
$info = $Q[16].o;
goto D_ADD_nonzero if $info is nonzero;
# Q = 0: result is Q1
```

```
        SET   t+2,Q             % MULT(copy(U),dV)
        PUSHJ t,:MULT
        SET   Q,t
        PUT   :rJ,warp
6H      JMP   ADD

DIV     LDOU  info,Q1,dINFOTYPE % compute U'/V
        BZ    info,7F           % if U' is not already 0
        GET   warp,:rJ

        LDOU  t,P2,dRLINK       % temp disconnect P2 from the tree
        STCO  1,P2,dRLINK
        SET   t+2,P2            % copy P2
        PUSHJ t+1,:COPYRTBT
        STOU  t,P2,dRLINK       % reconnect P2 into the tree

        SET   t+4,t+1           % assemble new Q1
        SET   t+3,Q1
        SET   t+2,N_DIV
        PUSHJ t+1,:TREE2

        SET   Q1,t+1            % finalize
        PUT   :rJ,warp

7H      LDOU  info,Q,dINFOTYPE  % compute (UV')/(V^2)
        BZ    info,7F           % if V' is not already 0
        GET   warp,:rJ

        LDOU  t,P1,dRLINK       % temp disconnect P1 from the tree
        STCO  1,P1,dRLINK
        SET   t+2,P1
        PUSHJ t+1,:COPYRTBT
        STOU  t,P1,dRLINK       % reconnect P1 into the tree

        SET   t+2,t+1           % compute (copy(P1)*Q)
        SET   t+1,Q
        PUSHJ t,:MULT           % don't touch t until Q <- ... !

        LDOU  t+1,P2,dRLINK     % temp disconnect P2 from the tree
        STCO  1,P2,dRLINK
        SET   t+3,P2            % copy P2
        PUSHJ t+2,:COPYRTBT
        STOU  t+1,P2,dRLINK     % reconnect P2 into the tree

        SET   t+4,N_CONST2      % build TREE(2)
        PUSHJ t+3,:TREE0

        SET   t+4,t+3           % assemble power
        SET   t+3,t+2
        SET   t+2,N_PWR
        PUSHJ t+1,:TREE2

        SET   t+3,t+1           % assemble new Q
        SET   t+2,t
        SET   t+1,N_DIV
        PUSHJ t,:TREE2

        SET   Q,t               % finalize
        PUT   :rJ,warp

7H      JMP   MINUS             % assemble (Q1 - Q)

PWR     LDOU  info,Q1,dINFOTYPE
        BZ    info,PWRQ         % no process for Q1, go to Q
        GET   warp,:rJ

        LDOU  t,P1,dRLINK       % temp disconnect P1
        STCO  1,P1,dRLINK
        SET   t+2,P1            % t+1 is R <- COPY(P1)
        PUSHJ t+1,:COPYRTBT
        STOU  t,P1,dRLINK       % reconnect P1

        LDOU  type,P2,dINFOTYPE
        AND   type,type,#F
        BNZ   type,PWRQ1B       % P2.TYPE=0 ?

PWRQ1A  LDOU  info,P2,dINFOTYPE
        SRU   info,info,4
        CMP   t,info,2
        BZ    t,PWRQ1X          % P2.TYPE=0 && P2.INFO!=2

        XOR   t,t,t
        NEGU  t,1
        ADDU  t,info,t          % t = P2.INFO - 1
        SLU   t,t,4             % reassemble INFO|TYPE

        SET   t+3,t
        PUSHJ t+2,:TREE0        % TREE(P2.INFO-1)

        SET   t+4,t+2
        SET   t+3,t+1           % R
        SET   t+2,#5E8          % "^"
        PUSHJ t+1,:TREE2        % keep [t+1] == R

        JMP   PWRQ1X            % R is ready for P2 constant and != 2

PWRQ1B  LDOU  t,P2,dRLINK       % temp disconnect P2
        STCO  1,P2,dRLINK
        SET   t+3,P2
        PUSHJ t+2,:COPYRTBT     % copied P2 in [t+2]
        STOU  t,P2,dRLINK       % reconnect P2

        SET   t+4,#10
        PUSHJ t+3,:TREE0        % built (1) in [t+3]

        SET   t+5,t+3
        SET   t+4,t+2
        SET   t+3,N_MINUS
        PUSHJ t+2,:TREE2        % assembled (copy P2)-(1) in [t+2]

        SET   t+4,t+2
        SET   t+3,t+1
        SET   t+2,N_PWR         % assembled in [t+1] == R
        PUSHJ t+1,:TREE2

PWRQ1X  LDOU  t,P2,dRLINK       % temp disconnect P2
```

```
$t = $Q;
$warp = $rJ;
$t = call AGC_GC($t);
$Q = $Q1;
$rJ = $warp;
goto D3;
D_ADD_nonzero:
$warp = $rJ;
$Q = call TREE2($N_PLUS, $Q1, $Q);
$rJ = $warp;
goto D3;

# MINUS: d/dx(u-v) = u' - v'
D_MINUS:
$info = $Q[16].o;
goto D_MINUS_vcheck if $info is nonzero;
# Q = 0: result is Q1
$t = $Q;
$warp = $rJ;
$t = call AGC_GC($t);
$Q = $Q1;
$rJ = $warp;
goto D3;
D_MINUS_vcheck:
$info = $Q1[16].o;
goto D_MINUS_both if $info is nonzero;
# Q1 = 0: result is -Q
$t = $Q1;
$warp = $rJ;
$t = call AGC_GC($t);
$Q = call TREE1($N_NEG, $Q);
$rJ = $warp;
goto D3;
D_MINUS_both:
$warp = $rJ;
$Q = call TREE2($N_MINUS, $Q1, $Q);
$rJ = $warp;
goto D3;

# TIMES: d/dx(u*v) = u'*v + u*v' (product rule)
D_TIMES:
$info = $Q1[16].o;
goto D_TIMES_Q if $info is zero;   # u' = 0
$warp = $rJ;
$t = $P2;
$t = call COPYRTBT($t);   # copy V
$Q1 = call MULT($Q1, $t);   # Q1 = u' * copy(V)
$rJ = $warp;
D_TIMES_Q:
$info = $Q[16].o;
goto D_ADD if $info is zero;   # v' = 0: just use Q1
$warp = $rJ;
$ret = [$P1].o;
[$P1].o = 1;
$t = $P1;
$t = call COPYRTBT($t);   # copy U while preserving the temporary bridge on P1
[$P1].o = $ret;
$Q = call MULT($t, $Q);   # Q = copy(U) * v'
$rJ = $warp;
goto D_ADD;

# DIV: d/dx(u/v) = (u'v - uv') / v^2
D_DIV:
$info = $Q1[16].o;
goto D_DIV_Q if $info is zero;
$warp = $rJ;
$ret = [$P2].o;
[$P2].o = 1;   # temp disconnect P2 from the tree
$t = $P2;
$t = call COPYRTBT($t);   # copy P2
[$P2].o = $ret;
$Q1 = call TREE2($N_DIV, $Q1, $t);
$rJ = $warp;
D_DIV_Q:
$info = $Q[16].o;
goto D_DIV_assemble if $info is zero;
$warp = $rJ;
$ret = [$P1].o;
[$P1].o = 1;
$t = $P1;
$t = call COPYRTBT($t);
[$P1].o = $ret;
$Q = call MULT($t, $Q);
$ret = [$P2].o;
[$P2].o = 1;
$t = $P2;
$t = call COPYRTBT($t);
[$P2].o = $ret;
$tmp = call TREE0($N_CONST2);
$t = call TREE2($N_PWR, $t, $tmp);
$Q = call TREE2($N_DIV, $Q, $t);
$rJ = $warp;
D_DIV_assemble:
goto D_MINUS;

# PWR: d/dx(u^v) - chain rule
D_PWR:
$info = $Q1[16].o;
goto D_PWR_Q if $info is zero;
$warp = $rJ;
$ret = [$P1].o;
[$P1].o = 1;
$t = $P1;
$t = call COPYRTBT($t);
[$P1].o = $ret;
$typev = $P2[16].o;
$typev = $typev & #F;
goto D_PWR_Q1_general if $typev is nonzero;
# v is constant
$info = $P2[16].o;
$info = $info >>u 4;
$tmp = $info - 2;
goto D_PWR_Q1_mul if $tmp is zero;
# build v-1 constant
$tmp = $info - 1;
$tmp = $tmp <<u 4;
```

```
        STCO  1,P2,dRLINK
        SET   t+3,P2
        PUSHJ t+2,:COPYRTBT      % copied P2
        STOU  t,P2,dRLINK        % reconnect P2

        SET   t+4,t+1            % R
        SET   t+3,t+2            % copy(P2)
        PUSHJ t+2,:MULT          % assembled copy(P2)*R and consume R

        % SET  t+2,t+2           % latest result is kept
        SET   t+1,Q1
        PUSHJ t,:MULT            % assembled Q1*<previous-assembly>

        SET   Q1,t               % finalize Q1

        PUT   :rJ,warp

PWRQ    LDOU  info,Q,dINFOTYPE
        BZ    info,PWRX          % no process required for Q, goto exit
        GET   warp,:rJ

        LDOU  t,P1,dRLINK
        STCO  1,P1,dRLINK
        SET   t+2,P1
        PUSHJ t+1,:COPYRTBT      % copied P1 in [t+1]
        STOU  t,P1,dRLINK

        SET   t+2,t+1
        SET   t+1,N_LN
        PUSHJ t,:TREE1           % [t+1] consumed

        SET   t+2,Q
        SET   t+1,t
        PUSHJ t,:MULT            % [t+1] consumed, [t] set

        LDOU  t+1,P1,dRLINK
        STCO  1,P1,dRLINK
        SET   t+3,P1
        PUSHJ t+2,:COPYRTBT      % copied P1 in [t+2]
        STOU  t+1,P1,dRLINK

        LDOU  t+1,P2,dRLINK
        STCO  1,P2,dRLINK
        SET   t+4,P2
        PUSHJ t+3,:COPYRTBT      % copied P2 in [t+3]
        STOU  t+1,P2,dRLINK

        SET   t+4,t+3
        SET   t+3,t+2
        SET   t+2,N_PWR
        PUSHJ t+1,:TREE2         % consumed [t+2] and [t+3], set [t+1]

        SET   t+3,t+1
        SET   t+2,t
        SET   t+1,N_TIMES
        PUSHJ t,:TREE2           % consumed [t] and [t+1]

        SET   Q,t                % finalize

        PUT   :rJ,warp

PWRX    JMP   ADD

        PREFIX :
```

```
  $tmp = call TREE0($tmp);
  $t = call TREE2($N_PWR, $t, $tmp);
  goto D_PWR_Q1_mul;
  D_PWR_Q1_general:
  # v is not constant: build copy(v)-1 and raise u to that power
  $ret = [$P2].o;
  [$P2].o = 1;
  $tmp = $P2;
  $tmp = call COPYRTBT($tmp);
  [$P2].o = $ret;
  $ret = call TREE0($N_CONST1);
  $tmp = call TREE2($N_MINUS, $tmp, $ret);
  $t = call TREE2($N_PWR, $t, $tmp);
  D_PWR_Q1_mul:
  $ret = [$P2].o;
  [$P2].o = 1;
  $tmp = $P2;
  $tmp = call COPYRTBT($tmp);
  [$P2].o = $ret;
  $t = call MULT($tmp, $t);
  $Q1 = call MULT($Q1, $t);
  $rJ = $warp;
  D_PWR_Q:
  $info = $Q[16].o;
  goto D_ADD if $info is zero;
  $warp = $rJ;
  $ret = [$P1].o;
  [$P1].o = 1;
  $t = $P1;
  $t = call COPYRTBT($t);
  [$P1].o = $ret;
  $t = call TREE1($N_LN, $t);
  $Q = call MULT($Q, $t);
  $ret = [$P1].o;
  [$P1].o = 1;
  $t = $P1;
  $t = call COPYRTBT($t);
  [$P1].o = $ret;
  $info = [$P2].o;
  [$P2].o = 1;
  $tmp = $P2;
  $tmp = call COPYRTBT($tmp);
  [$P2].o = $info;
  $t = call TREE2($N_PWR, $t, $tmp);
  $Q = call TREE2($N_TIMES, $Q, $t);
  $rJ = $warp;
  goto D_ADD;
end

def AllocNode($info) -> $node:
  if $NodeFree != 0:
    $node = $NodeFree;
    $NodeFree = [$node].o;
  else:
    if $NodeNext == 0:
      $NodeNext = $NodePool;
    end
    $node = $NodeNext;
    $NodeNext = $NodeNext + 24;
  end

  [$node].o = 0;
  $node[8].o = 0;
  $node[16].o = $info;
  return $node;
end

def TREE0($info) -> $node:
  if $NodeFree != 0:
    $node = $NodeFree;
    $NodeFree = [$node].o;
  else:
    if $NodeNext == 0:
      $NodeNext = $NodePool;
    end
    $node = $NodeNext;
    $NodeNext = $NodeNext + 24;
  end

  [$node].o = 0;
  $node[8].o = 0;
  $node[16].o = $info;
  return $node;
end

def TREE1($typev, $child) -> $node:
  register $info;
  register $savedChild;

  $info = $typev;
  $savedChild = $child;

  if $NodeFree != 0:
    $node = $NodeFree;
    $NodeFree = [$node].o;
  else:
    if $NodeNext == 0:
      $NodeNext = $NodePool;
    end
    $node = $NodeNext;
    $NodeNext = $NodeNext + 24;
  end

  [$node].o = 0;
  $node[8].o = 0;
  $node[16].o = $info;
  $node[8].o = $savedChild;
  return $node;
end

def TREE2($typev, $left, $right) -> $node:
  register $info;
  register $savedLeft;
  register $savedRight;

  $info = $typev;
```

```
  $savedLeft = $left;
  $savedRight = $right;

  if $NodeFree != 0:
    $node = $NodeFree;
    $NodeFree = [$node].o;
  else:
    if $NodeNext == 0:
      $NodeNext = $NodePool;
    end
    $node = $NodeNext;
    $NodeNext = $NodeNext + 24;
  end

  [$node].o = 0;
  $node[8].o = 0;
  $node[16].o = $info;
  $node[8].o = $savedLeft;
  [$node].o = $savedRight;
  return $node;
end

def COPYRTBT($node) -> $copy:
  register $src;
  register $left;
  register $right;
  register $link;
  register $info;
  register $tmp;
  register $zero;
  register $warp;

  if $node == 0:
    $zero = 0;
    return $zero;
  end

  $src = $node;
  $info = $src[16].o;
  $warp = $rJ;
  $copy = call AllocNode($info);
  $rJ = $warp;

  $left = $src[8].o;
  if $left != 0:
    $warp = $rJ;
    $tmp = call COPYRTBT($left);
    $rJ = $warp;
    $copy[8].o = $tmp;
  end

  $link = [$src].o;
  $tmp = $link & 1;
  if $tmp is even:
    if $link != 0:
      $warp = $rJ;
      $tmp = call COPYRTBT($link);
      $rJ = $warp;
      [$copy].o = $tmp;
    end
  end

  return $copy;
end

def MULT($left, $right) -> $node:
  register $savedLeft;
  register $savedRight;

  $savedLeft = $left;
  $savedRight = $right;

  if $NodeFree != 0:
    $node = $NodeFree;
    $NodeFree = [$node].o;
  else:
    if $NodeNext == 0:
      $NodeNext = $NodePool;
    end
    $node = $NodeNext;
    $NodeNext = $NodeNext + 24;
  end

  [$node].o = 0;
  $node[8].o = 0;
  $node[16].o = $N_TIMES;
  $node[8].o = $savedLeft;
  [$node].o = $savedRight;
  return $node;
end

def AGC_GC($node) -> $zero:
  register $curr;
  register $left;
  register $right;
  register $tmp;
  register $warp;

  if $node == 0:
    $zero = 0;
    return $zero;
  end

  $curr = $node;
  $left = $curr[8].o;
  if $left != 0:
    $warp = $rJ;
    call AGC_GC($left);
    $rJ = $warp;
  end

  $right = [$curr].o;
  $tmp = $right & 1;
  if $tmp is even:
    if $right != 0:
      $warp = $rJ;
```

```
      call AGC_GC($right);
      $rJ = $warp;
    end
  end

  [$curr].o = $NodeFree;
  $curr[8].o = 0;
  $curr[16].o = 0;
  $NodeFree = $curr;
  $zero = 0;
  return $zero;
end
```

TAOCP 3.2.1.1 – Linear Congruential Method exercises

ex2: Linear congruential random number generator

ex2.mms

```
<<<---
% Author M.Ruckert
 LOC #100
Main IS @
--->>>
 PREFIX :Random:
a GREG
X GREG
c IS ...

:Random MULU X,X,a$ X is a \cdot X mod w$
 INCL  X,c    $ X \is ( a \cdot X + c ) \bmod w$
 SET  $0,X
 POP  1,0
```

ex2.plm

```
# Linear congruential random number generator (TAOCP 3.2.1.1, Exe...
# X_{n+1} = ($a * X_n + c) mod 2^64

# Based on an .mms file by M.Ruckert
greg $a;
greg $X;
const c = 2009;

def Random():
  $X = $X *u $a;
  $X = $X +u c;
  return $X;
end

# Main: initialize $a and $X, then call Random
$a = 6364136223846793005;
$X = 1;
$0 = call Random();
trap 0, HALT, 0;
```

ex5ab: Modular add and subtract (unsigned)

ex5ab.mms

```
<<<---
% Author M.Ruckert
 LOC #100
Main IS @
--->>>
x IS $0
y IS $1
m IS $2
diff IS $3
sum IS $4
tmp IS $5

 SUBU diff,x,y
 CMPU tmp,x,y
 ZSN tmp,tmp,m
        ADDU diff,diff,tmp

 SUBU tmp,m,y
 SUBU sum,x,tmp
 CMPU tmp,x,tmp
 ZSN tmp,tmp,m
 ADDU sum,sum,tmp
```

ex5ab.plm

```
# Unsigned modular subtraction and addition (TAOCP 3.2.1.1, Exerc...
# x, y, m in $0, $1, $2; results: diff in $3, sum in $4

# Based on an .mms file by M.Ruckert
register $x, $y, $m, $diff, $sum, $tmp;

# Modular subtraction: diff = (x - y) mod m
$diff = $x -u $y;
$tmp = $x cmpu $y;
$tmp ?= $m if $tmp is neg;
$diff = $diff +u $tmp;

# Modular addition: sum = (x + y) mod m
$tmp = $m -u $y;
$sum = $x -u $tmp;
$tmp = $x cmpu $tmp;
$tmp ?= $m if $tmp is neg;
$sum = $sum +u $tmp;

trap 0, HALT, 0;
```

ex5c: Modular add and subtract (signed)

ex5c.mms

```
<<<---
% Author M.Ruckert
 LOC #100
Main IS @
--->>>
x IS $0
y IS $1
m IS $2
diff IS $3
sum IS $4
tmp IS $5

 SUB diff,x,y
 ZSN tmp,diff,m
 ADD diff,diff,tmp

 SUB tmp,m,y
 SUB sum,x,tmp
 ZSN tmp,sum,m
 ADD sum,sum,tmp
```

ex5c.plm

```
# Signed modular subtraction and addition (TAOCP 3.2.1.1, Exercis...
# x, y, m in $0, $1, $2; results: diff in $3, sum in $4

# Based on an .mms file by M.Ruckert
register $x, $y, $m, $diff, $sum, $tmp;

# Modular subtraction: diff = (x - y) mod m
$diff = $x - $y;
$tmp ?= $m if $diff is neg;
$diff = $diff + $tmp;

# Modular addition: sum = (x + y) mod m
$tmp = $m - $y;
$sum = $x - $tmp;
$tmp ?= $m if $sum is neg;
$sum = $sum + $tmp;

trap 0, HALT, 0;
```

ex8: Efficient multiplication aX mod w

ex8.mms

```
<<<---
% Author M.Ruckert
 LOC #100
Main IS @
--->>>
a GREG
X GREG
c GREG
q IS $0
r IS $1
tmp IS $2
        MULU r,a,X; GET q,rH         $ (q,r) \is aX = qw+r$
 NOR tmp,q,0           $ tmp \is (w-1)-q $
 SUBU X,r,tmp                $ X \is r+q-(w-1) \bmod w$
        CMPU tmp,r,tmp; BNN tmp,Exit Exit if $r \ge (w-1)-q$
 SUBU X,X,1                  $ X \is r+q+(w-1) \bmod w$
<<<---
Exit TRAP 0,Halt,0
--->>>
```

ex8.plm

```
# RNG with correction for near-overflow (TAOCP 3.2.1.1, Exercise ...
# X_{n+1} = ($a * X_n) mod (2^64 - 1)  [approximately]

# Based on an .mms file by M.Ruckert
greg $a;
greg $X;
greg $c;
register $q, $r, $tmp;

def Random():
  $r = $a *u $X;
  $q = $rH;
  # NOR: tmp = ~(q | 0) = ~q  [NOR $x,$y,0 = NOT $y]
  $tmp = $q nor 0;   # NOR tmp,q,0: tmp = (2^64-1) - q
  $X = $r -u $tmp;   # $X = r + q mod (2^64-1) approx
  $tmp = $r cmpu $tmp;
  goto Done if $tmp is nonneg;
  $X = $X -u 1;
  Done:
  return $X;
end

$a = 6364136223846793005;
$c = 0;
$X = 1;
$0 = call Random();
trap 0, HALT, 0;
```

TAOCP 3.2.2 – Other Methods (Program A)

proga1: Program A – Additive (Fibonacci) generator

proga1.mms

```
<<<---
% Author M.Ruckert
 LOC Data_Segment
X OCTA 1,2,3,4,5,6,7,8,9,10

 LOC 100
Main IS @
--->>>
%%X OCTA  %$X_{54},...,X_0$
j GREG 24*8
k GREG 55*8
Y GREG X-8 $Y is Address of Y[0]$

result IS $0
tmp IS $1

 LDOU result,Y,j  A1. Add. $result is Y[j]$
 LDOU tmp,Y,k   $\.{tmp} \is Y[j]$
 ADDU result,result,tmp $\.{result} \is Y[k] + Y[j]$
 STOU result,Y,k  $Y[k] \is Y[k] + Y[j]$
 SUB j,j,8   A2. Advance. $j is j-1$
 SUB k,k,8   $k \is k-1$
 SET tmp,55*8
 CSNP j,j,tmp   If $j \le 0$, set $j \is 55$
 CSNP k,k,tmp   If $k \le 0$, set $k \is 55$
```

proga1.plm

```
# Additive RNG step using a circular array (TAOCP 3.2.2, Algorith...
# X[k] = (X[k] + X[j]) mod 2^64  (linear congruential, additive m...
# j and k are byte-offsets into a circular buffer; decremented ea...

# Based on an .mms file by M.Ruckert
const J0 = 24*8;
const K0 = 55*8;

segment data $DataBase;
octa X 1, 2, 3, 4, 5, 6, 7, 8, 9, 10;
segment code;

register $Y = $0;# Base address of X[0] adjusted for offset
register $result, $tmp;
greg $j = J0;
greg $k = K0;

# Initialize: Y = $DataBase - 8  (address of X[-1] so X[0] = Y+8)
$Y = $DataBase - 8;

# Step A1: Add.  result = X[k] + X[j]

$result = $Y[$j].o;   # result = X[j]
$tmp    = $Y[$k].o;   # tmp    = X[k]
$result = $result +u $tmp;   # result = X[k] + X[j]
$Y[$k].o = $result;   # X[k] = X[k] + X[j]

# Step A2: Advance.  j = j-1 (wrap 0 -> 55), k = k-1 (wrap 0 -> 5...
$j = $j - 8;
$k = $k - 8;
$tmp = K0;
$j = $tmp if $j is nonpos;
$k = $tmp if $k is nonpos;

trap 0, HALT, 0;
```

proga2: Program A – Additive generator (alternative)

proga2.mms

```
<<<---
% Author M.Ruckert
 LOC #100
Main IS @
--->>>
ELEMENTS IS 64  Number of elements in circular buffer
SIZE  IS ELEMENTS*8 Size of buffer in BYTE
MASK  IS SIZE-1  Mask to facilitate computations modulo \.{SIZE}
  LOC (@+SIZE-1)&-MASK    Align location to a multiple of \.{SIZE}
X  OCTA ...  $X_0,...,X_{54}$
  LOC X+SIZE  Reserve SIZE byte
Xn  GREG X+54*8          $\.{Xn} \is Address of current X_n = X_{54}$

result IS $0
base IS $1
offset IS $2
mask IS $3
tmp IS $4

        SET mask,MASK
 ANDN base,Xn,mask   Extract \.{base} and \.{offset} from \.{Xn}
 AND offset,Xn,mask   offset of $X_{n}$
 ADD offset,offset,8   A2. Advance. $n is n+1$
 AND offset,offset,mask
 ADDU Xn,base,offset   $\.{Xn} \is Address of X_{n+1}$
 SUB offset,offset,24*8  offset of $X_{n-24}$
 AND offset,offset,mask
 LDOU result,base,offset  $\.{result} \is  X_{n-24}$
 SUB offset,offset,(55-24)*8         \.{offset} of $X_{n-55}$
 AND offset,offset,mask
 LDOU tmp,base,offset   $\.{tmp} \is  X_{n-55}$
 ADDU result,result,tmp  A1. Add.
 STOU result,Xn,0    $X_n \is X_{n-24}+X_{n-55} \bmod m$
```

proga2.plm

```
# Additive RNG with power-of-2 circular buffer (TAOCP 3.2.2, Algo...
# Uses bitwise AND for modular indexing (buffer size must be powe...
# X[n+1] = X[n-24] + X[n-55] mod 2^64

# Based on an .mms file by M.Ruckert
const ELEMENTS = 64;
const SIZE = ELEMENTS*8;
const MASK = SIZE-1;

segment data;
# Xbuf must be SIZE-aligned for the ANDN/AND base+offset trick to...
# segment data starts at #2000000000000000 which is already align...
reserve Xbuf SIZE;   # X_0..X_63: SIZE bytes, zero-initialised

segment code;
register $result, $base, $offset, $mask, $tmp;
greg $Xn = Xbuf + 54*8;   # address of current X_n = X_54

$mask = MASK;

# Step A2: Advance to next n (n = n+1)
$base   = $Xn andn $mask;   # base = Xn & ~MASK  (aligned block)
$offset = $Xn & $mask;   # offset within block
$offset = $offset + 8;   # advance by 1 element
$offset = $offset & $mask;   # wrap modulo SIZE
$Xn     = $base +u $offset;   # new Xn

# Step A1: Load X[n-24] and X[n-55], add them, store X[n]
$offset = $offset - 192;   # 24*8 = 192
$offset = $offset & $mask;
$result = $base[$offset].o;   # result = X[n-24]

$offset = $offset - 248;   # (55-24)*8 = 248
$offset = $offset & $mask;
$tmp    = $base[$offset].o;   # tmp = X[n-55]

$result = $result +u $tmp;   # result = X[n-24] + X[n-55]
[$Xn].o = $result;   # X[n] = result

trap 0, HALT, 0;
```

TAOCP 3.6 – Random Number Summary

ex1: Uniform random integer in range [0,k)

ex1.mms

```
<<<---
% Author M.Ruckert
 LOC #100
Main IS @
--->>>
 PREFIX :RandI:
X GREG 1  $X_0 is 1$
a IS      6364136223846793005 (see Section 3.3.4, Table 1, line 26)
c IS 2009
k IS $0  Parameter
tmp IS $1

:RandI SETH tmp,(a>>48)&#FFFF load constant \.{a}
 INCMH tmp,(a>>32)&#FFFF
 INCML tmp,(a>>16)&#FFFF
 INCL tmp,a&#FFFF
 MULU X,X,tmp  $ X \is a \cdot X $
 INCL    X,c      $ X \is ( a \cdot X + c ) \bmod m$
 MULU tmp,X,k  $ (\.{rH},\.{tmp}) \is Xk $
 GET $0,:rH  $Return value is \lfloor Xk/m \rfloor$.
 POP  1,0
```

ex1.plm

```
# RNG using full 64-bit multiplier loaded via wyde-immediate ops ...
# X_{n+1} = (a * X_n + c) mod 2^64
# a = 6364136223846793005  (line 26 of Table 1, Section 3.3.4)
# c = 2009

# Based on an .mms file by M.Ruckert
greg $X = 1;
register $k, $tmp;
const c = 2009;

def RandI($0):
  # Load 64-bit constant a into tmp using wyde-immediate ops
  $tmp = 0;
  $tmp orh=  #5851;   # high wyde of a
  $tmp ormh= #F42D;   # med-high wyde
  $tmp orml= #4C95;   # med-low wyde
  $tmp orl=  #7D4D;   # low wyde of a
  $X = $X *u $tmp;    # $X = a * $X
  $X = $X +u c;       # $X = a*$X + c
  $tmp = $X *u $k;    # ($rH,tmp) = $X*k
  $0 = $rH;   # return floor($X*k / 2^64)
  return $0;
end

$1 = 3;
$0 = call RandI($1);
trap 0, HALT, 0;
```

TAOCP 4.2.1 – Floating-Point Notation (Program A)

fix: Convert floating-point to integer

fix.mms

```
<<<---
% Author M.Ruckert
 LOC #100
--->>>
 PREFIX :Fix:
u IS $0 1. parameter and return value
s IS $1 $u$ unpacked
e IS $2
f IS $3
carry IS $4 for rounding
tmp IS $5

:Fix ZSN s,u,1    % Unpack. Set sign
        SETH tmp,#FFF0; ANDN f,u,tmp  Remove sign and exponet bits to get $f$
 INCH f,#0010 Add hidden bit.  $2^{52}\le f < 2^{53}$
 SLU e,u,1; SRU e,e,53  Get exponent. Now $u=f2^{e-q}/2^{52}$.
             % Check exponent range.
 SET tmp,1023-1; SUB e,e,tmp  Subtract $q-1$. Now $u = f2^e/2^{53}$
 BN e,Zero    $u<1/2$
 CMP tmp,e,63; BP tmp,:Error  $u\ge2^{63}$ (Overflow)
 SUB e,e,53          Now $u=f2^e$.
 BN e,Round
 SLU u,f,e
 JMP Sign
     % Round.
Round NEG e,e; SUB e,e,2; SRU f,f,e Now $u=f/4$
 SRU u,f,2      $u \is \lfloor f \rfloor$
 AND tmp,f,3; CMP tmp,tmp,2  Compare $(f \bmod 4)/4$ to $1/2$
 CSOD carry,u,1   $u$ is odd; round up if $(f \bmod 4)/4 \ge {1\over 2}$
 CSEV carry,u,tmp $u$ is even; round up if $(f \bmod 4)/4 > {1\over 2}$
 ZSNN carry,tmp,carry Round down if $(f \bmod 4)/4 < {1\over 2}$
 ADDU u,u,carry
     % Attach Sign.
Sign NEG     tmp,u; CSNZ u,s,tmp         If $s$, set $u \is -u$.
 POP 1,0

Zero SET u,0    % Return zero.
 POP 1,0
<<<---
    PREFIX :
Main    FLOT    $1,77
          PUSHJ   $0,Fix
    NEG    $1,33
    FLOT    $1,$1
    PUSHJ   $0,Fix
    FLOT    $2,2
    FLOT    $1,1
    FDIV    $1,$1,$2
    FIX    $0,$1
    PUSHJ   $0,Fix
    FLOT    $2,2
    FLOT    $1,3
    FDIV    $1,$1,$2
    FIX    $0,$1
    PUSHJ   $0,Fix

    SET    $255,0
    TRAP    0,Halt,0

Error    SET    $255,1
    TRAP    0,Halt,0
--->>>
```

fix.plm

```
# Convert IEEE 754 double to integer (TAOCP 4.2.1, Algorithm Fix)
# Software implementation: rounds to nearest even.
# Parameter: $0 = float (IEEE 754 double encoding as 64-bit integ...
# Return: $0 = integer result

# Based on an .mms file by M.Ruckert
def Fix($u):
  register $s;# sign bit: 0=positive, 1=negative
  register $e;# exponent (biased, then adjusted)
  register $f;# significand with hidden bit
  register $carry;# rounding carry
  register $tmp;

  # Unpack
  $s ?= 1 if $u is neg;   # ZSN: s = 1 if u < 0 (sign bit set)
  $tmp = #FFF0000000000000;
  $f = $u andn $tmp;   # ANDN: remove sign and exponent bits
  $f = $f | #0010000000000000;   # add hidden bit: 2^52 <= f < 2^53
  $e = $u <<u 1;
  $e = $e >>u 53;   # biased exponent

  # Check exponent range; subtract q-1 = 1022
  $e = $e - 1022;   # now u = f * 2^e / 2^53
  goto FixZero if $e is neg;
  $tmp = $e - 63;
  goto FixError if $tmp is pos;   # |u| >= 2^63: overflow
  $e = $e - 53;   # now u = f * 2^e
  goto Round if $e is neg;
  $u = $f <<u $e;
  goto FixSign;

  Round:
  $tmp = - $e;
  $tmp = $tmp - 2;
  $f = $f >>u $tmp;   # f = original_f / 4
  $u = $f >>u 2;   # u = floor(f/4)
  $tmp = $f & 3;
  $tmp = $tmp - 2;   # compare (f mod 4) to 2
  $carry = 1 if $u is odd;   # CSOD: round up if u odd and f mod 4 >= 2
  $carry = $tmp if $u is even;   # CSEV: round up if u even and f mod 4 > 2
  $carry ?= 0 if $tmp is neg;   # ZSNN: zero carry if f mod 4 < 2
  $u = $u +u $carry;

  FixSign:
  $tmp = - $u;
  $u = $tmp if $s is nonzero;   # CSNZ: negate if negative
  return $u;

  FixZero:
  $u = 0;
  return $u;

  FixError:
  trap 0, 1, 0;
  return $u;
end

segment code;
register $r, $a, $b, $tmp;

$b = 77;
$a = flot. $b;
$r = call Fix($a);

$tmp = -33;
$a = flot. $tmp;
$r = call Fix($a);

$b = 2;
$tmp = flot. $b;
$b = 1;
$a = flot. $b;
$a = $a /. $tmp;   # 1.0 / 2.0 = 0.5
$r = fix. $a;   # hardware FIX for comparison
$r = call Fix($a);

$b = 2;
$tmp = flot. $b;
$b = 3;
$a = flot. $b;
$a = $a /. $tmp;   # 3.0 / 2.0 = 1.5
$r = fix. $a;
$r = call Fix($a);

trap 0, HALT, 0;
```

fmod: Floating-point modulo

fmod.mms

```
<<<---
% Author M.Ruckert
 LOC #100

w IS $0 return value
u IS $0 1. parameter
s IS $1 $u$ unpacked
e IS $2
f IS $3
f0 IS $4
carry IS $5 for rounding
tmp IS $6
--->>>
Fmod ZSN s,u,1    %1. Unpack. Set sign.
       SETH tmp,#FFF0; ANDN f,u,tmp  Remove sign and exponent from $f$.
 INCH f,#0010       Add hidden bit.
 SLU e,u,1; SRU e,e,53  Get exponent.
 SET f0,0           $u=\pm (f,f_0)2^{e-q}/2^{52}$.
 SET tmp,1023; SUB e,e,tmp  %2. Subtract $q$.
 BNP e,0F           Branch if $u$ has no integer part.
 ADD tmp,e,12; SLU f,f,tmp  %3. Remove integer part.
 SRU f,f,12
 SET e,0
0H BZ s,0F
 ADD tmp,e,64; SLU f0,f,tmp  %4. Complement fraction part.
 NEG tmp,e; SRU f,f,tmp  $(f,f_0)\is (f,0)/2^e$
 SET e,0            $e \is 0$
 NEGU f0,f0
 ZSNZ carry,f0,1; ADDU f,f,carry
 SETH tmp,#0010; SUBU f,tmp,f $(f,f_0) is 1 - (f,f_0)$
 SET s,0    $(f,f_0)>0$
0H INCL e,1023    %5. Add $q$.
 OR tmp,f,f0; BNZ tmp,Normalize  %6. Normalize, if not zero
 SET w,0            else $w\is 0$
<<<---
 POP 1,0

Normalize  SRU tmp,f,53; BP tmp,4F N1. Test $f$. If $f\ge 2$, scale right.
    OR tmp,f,f0; BZ tmp,Error underflow

2H    SRU tmp,f,52; BP tmp,5F N2. Is $f$ normalized?

    ZSN carry,f0,1; SLU f0,f0,1 N3. Scale left.
    SLU f,f,1; ADDU f,f,carry
    SUB e,e,1
    JMP 2B

4H    AND  carry,f,1 N4. Scale right.
    SLU carry,carry,63
    SRU f0,f0,1; ADDU f0,f0,carry
    SRU f,f,1
    ADD e,e,1

5H    SETH tmp,#8000 N5. Round.
    CMPU tmp,f0,tmp Compare $f_0$ to ${1\over 2}$
    CSOD carry,f,1   $f$ is odd. Round up if $f_0 \ge {1\over 2}$
    CSEV carry,f,tmp $f$ is even. Round up if $f_0 > {1\over 2}$
    ZSNN carry,tmp,carry Round down if $f_0 < {1\over 2}$
    ADDU f,f,carry
    SET f0,0
    SRU tmp,f,53; BP tmp,4B  Rounding overflow

    SET tmp,#7FE; CMP tmp,e,tmp    N6. Check $e$.
    BP tmp,Error Overflow
    BNP e,Error  Underflow

    SLU w,s,63  N5. Pack.
    SLU tmp,e,52; OR u,u,tmp
    ANDNH f,#FFF0 remove hidden bit
    OR w,w,f

    SET $0,u
    POP 1,0

Error    SET    $255,1
    TRAP    0,:Halt,0

    PREFIX :
Main    FLOT    $255,10
    FLOT    $1,77
    FDIV    $1,$1,$255
          PUSHJ   $0,Fmod
    NEG    $1,33
    FLOT    $1,$1
    FDIV    $1,$1,$255
    PUSHJ   $0,Fmod
    NEG    $1,33
    FLOT    $1,$1
    FLOT    $255,100
    FDIV    $1,$1,$255
    FDIV    $1,$1,$255
    PUSHJ   $0,Fmod
    FMUL    $255,$255,$255; FMUL $255,$255,$255; FMUL $255,$255,$255

    SET    $1,$255
    PUSHJ   $0,Fmod

    FLOT    $1,1
    FDIV    $1,$1,$255
    PUSHJ   $0,Fmod

    NEG    $1,1
    FLOT    $1,$1
    FDIV    $1,$1,$255
    PUSHJ   $0,Fmod

    NEG    $1,2
    FLOT    $1,$1
    FDIV    $1,$1,$255
    PUSHJ   $0,Fmod
```

fmod.plm

```
# Float modulo (fractional part): return fmod(u, 1.0) (TAOCP 4.2....
# Returns the fractional part of |u|, with original sign.
# Also demonstrates normalization with (f, f0) 128-bit extended p...
# Parameters: $0 = u (IEEE 754 double)
# Return: $0 = w = fractional part

# Based on an .mms file by M.Ruckert
def Fmod($w):
  register $u = $0;# 1st parameter (same register as w)
  register $s = $1;# sign of u
  register $e = $2;# exponent
  register $f = $3;# high 64 bits of significand
  register $f0 = $4;# low 64 bits
  register $carry, $tmp;

  # 1. Unpack
  $s = 0;
  $s ?= 1 if $u is neg;   # ZSN: sign bit
  $tmp = #FFF0000000000000;
  $f = $u andn $tmp;   # remove sign and exponent bits
  $f = $f | #0010000000000000;   # add hidden bit
  $f0 = 0;
  $e = $u <<u 1;
  $e = $u <<u 1;
  $e = $e >>u 53;   # correct: 11-bit biased exponent
  $e = $e - 1023;   # 2. Subtract q (unbias)

  goto FmodNoIntPart if $e is nonpos;

  # 3. Remove integer part: shift out integer bits
  $tmp = $e + 12;
  $f = $f <<u $tmp;
  $f = $f >>u 12;   # f now holds only fractional bits (2^52 scale)
  $e = 0;

  FmodNoIntPart:
  goto FmodNormalize if $s is zero;

  # 4. Complement fraction (for negative u)
  $tmp = $e + 64;   # = 64 + e; since e <= 0, this is <= 64
  $f0 = $f <<u $tmp;   # f0 = f << (64+e) - low bits shifted out
  $tmp = - $e;
  $f = $f >>u $tmp;   # f >>= -e
  $e = 0;
  $f0 = -u $f0;
  $carry = 0;
  $carry ?= 1 if $f0 is nonzero;
  $f = $f +u $carry;
  $tmp = #0010000000000000;
  $f = $tmp -u $f;   # f = 2^52 - f (complement)
  $s = 0;

  FmodNormalize:
  $e = $e + 1023;   # 5. Add q
  $tmp = $f | $f0;
  goto FmodNorm1 if $tmp is nonzero;
  $w = 0;
  return $w;

  FmodNorm1:
  # N1. Test f
  $tmp = $f >>u 53;
  goto FmodN4 if $tmp is pos;
  $tmp = $f | $f0;
  goto FmodError if $tmp is zero;   # underflow

  FmodN2:
  $tmp = $f >>u 52;
  goto FmodN5 if $tmp is pos;

  # N3. Scale left
  $carry = 0;
  $carry ?= 1 if $f0 is neg;
  $f0 = $f0 <<u 1;
  $f = $f <<u 1;
  $f = $f +u $carry;
  $e = $e - 1;
  goto FmodN2;

  FmodN4:
  # N4. Scale right
  $carry = $f & 1;
  $carry = $carry <<u 63;
  $f0 = $f0 >>u 1;
  $f0 = $f0 +u $carry;
  $f = $f >>u 1;
  $e = $e + 1;
  # fall through to N5

  FmodN5:
  # N5. Round (omitted in this version)
  # N6. Check e
  $tmp = #7FE;
  $tmp = $tmp - $e;
  goto FmodError if $tmp is neg;   # overflow
  goto FmodError if $e is nonpos;   # underflow

  # N7. Pack
  $w = $s <<u 63;
  $tmp = $e <<u 52;
  $w = $w | $tmp;
  $f andnh= #FFF0;   # remove hidden bit from f
  $w = $w | $f;
  return $w;

  FmodError:
  trap 0, 1, 0;
  $w = 0;
  return $w;
end
```

```
    SETH    $1,#0010 very small positiv
    PUSHJ   $0,Fmod

    SETH    $1,#8010 very small negativ
    PUSHJ   $0,Fmod

    SET    $255,0
    TRAP    0,Halt,0

--->>>
```

```
segment code;
register $a, $b;
register $r = $0;
register $ten = $3;

$ten = 10;
$ten = flot. $ten;
$b = 77;
$a = flot. $b;
$a = $a /. $ten;   # 7.7
$r = call Fmod($a);

$b = 33;
$b = - $b;
$a = flot. $b;
$a = $a /. $ten;   # -3.3
$r = call Fmod($a);

trap 0, HALT, 0;
```

proga: Program A – Floating-point addition and subtraction

proga.mms

```
<<<---
% Author M.Ruckert
 LOC #100
w IS $0 return value
u IS $0 1. parameter
v IS $1 2. parameter

s IS $2 $w$ unpacked
e IS $3
f IS $4
f0 IS $5 The low 64 bit of $f$
carry IS $13 The shuttle between $f$ and $f_0$.

su IS $6 $u$ unpacked
eu IS $7
fu IS $8

sv IS $9 $v$ unpacked
ev IS $10
fv IS $11

d IS $12 $e_u - e_v$
tmp IS $14

--->>>
Fsub SETH tmp,#8000; XOR v,v,tmp  Change sign of operand.
Fadd SLU eu,u,1; SLU ev,v,1  Remove sign bit.
 CMPU tmp,eu,ev   A2. Assume $e_u \ge e_v$.
 BNN tmp,0F           Branch if $ (e_u, f_u) \ge (e_v, f_v) $
 SET tmp,u; SET u,v; SET v,tmp       else swap $u$ and $v$
 SLU eu,u,1; SLU ev,v,1         and remove sign bit again.

0H SRU eu,eu,53; SRU ev,ev,53 A1. Unpack.
        SETH tmp,#FFF0   Get sign and exponent mask.
 ANDN fu,u,tmp; ANDN fv,v,tmp  Remove sign and exponent bits.
 INCH fu,#0010; INCH fv,#0010  Add hidden bit.
 SRU su,u,63; SRU sv,v,63  Get sign bit.

 SET e,eu; SET s,su  A3. Set $e_w is e_u$.

 SUB d,eu,ev  A4. Test $e_u-e_v$.
 CMP tmp,d,53+2; BN tmp,5F $e_u-e_v \ge p+2$?
 SET f,fu; SET f0,0  $(f,f_0) \is f_u$
 JMP 7F

5H     NEG tmp,64,d A5. Scale right.
 SLU f0,fv,tmp Shift $(f_v,f_0)$ to the right $e_u-e_v$ places
 SRU fv,fv,d

 CMP tmp,su,sv; BNZ tmp,0F Signs $s_u$ and $s_v$ differ.
 ADDU f,fu,fv A6. Add.
 JMP 7F
0H NEGU f0,f0; ZSNZ carry,f0,1  A6. Subtract.
 SUBU f,fu,fv; SUBU f,f,carry
 OR tmp,f,f0; BZ tmp,Error  Result is zero.
7H IS @   A7. Normalize.
Normalize  SRU tmp,f,53; BP tmp,4F N1. Test $f$. If $f\ge 2$, scale right.
    OR tmp,f,f0; BZ tmp,Error underflow

2H    SRU tmp,f,52; BP tmp,5F N2. Is $f$ normalized?

    ZSN carry,f0,1; SLU f0,f0,1 N3. Scale left.
    SLU f,f,1; ADDU f,f,carry
    SUB e,e,1
    JMP 2B

4H    AND  carry,f,1 N4. Scale right.
    SLU carry,carry,63
    SRU f0,f0,1; ADDU f0,f0,carry
    SRU f,f,1
    ADD e,e,1

5H    SETH tmp,#8000 N5. Round.
    CMPU tmp,f0,tmp Compare $f_0$ to ${1\over 2}$
    CSOD carry,f,1   $f$ is odd. Round up if $f_0 \ge {1\over 2}$
    CSEV carry,f,tmp $f$ is even. Round up if $f_0 > {1\over 2}$
    ZSNN carry,tmp,carry Round down if $f_0 < {1\over 2}$
    ADDU f,f,carry
    SET f0,0
    SRU tmp,f,53; BP tmp,4B  Rounding overflow

    SET tmp,#7FE; CMP tmp,e,tmp    N6. Check $e$.
    BP tmp,Error Overflow
    BNP e,Error  Underflow

    SLU w,s,63  N7. Pack.
    SLU tmp,e,52; OR w,w,tmp
    ANDNH f,#FFF0 remove hidden bit
    OR w,w,f
<<<---
    SET $0,w
    POP 1,0

Main    NEG    $2,100
    FLOT    $2,$2
    FLOT    $3,103
    PUSHJ   $1,Fadd
%    FLOT    $2,
%    PUSHJ   $0,Fsub
    SET    $255,0
    TRAP    0,Halt,0

Error    SET    $255,1
    TRAP    0,Halt,0
--->>>
```

proga.plm

```
# Software floating-point addition and subtraction (TAOCP 4.2.1, ...
# Fadd($0, $1): w = u + v (IEEE 754 double, software)
# Fsub($0, $1): w = u - v (negates v then calls Fadd)
# Return value in $0.

# Based on an .mms file by M.Ruckert
def Fsub($u, $v):
  register $tmp = $14;
  $tmp = #8000000000000000;
  $v = $v ^ $tmp;   # change sign of v
  # fall through to Fadd body via goto
  goto FaddBody;
end

def Fadd($u, $v):
  FaddBody:
  register $w = $0;# return value
  register $s = $2;# sign of result
  register $e = $3;# exponent of result
  register $f = $4;# significand (high)
  register $f0 = $5;# significand (low)
  register $carry = $13;
  register $su = $6;# u unpacked
  register $eu, $fu;
  register $sv = $9;# v unpacked
  register $ev, $fv;
  register $d = $12;# eu - ev
  register $tmp = $14;

  # A2. Assume eu >= ev; swap if needed
  $eu = $u <<u 1;
  $ev = $v <<u 1;
  $tmp = $eu cmpu $ev;
  goto FaddNoSwap if $tmp is nonneg;
  $tmp = $u; $u = $v; $v = $tmp;
  $eu = $u <<u 1;
  $ev = $v <<u 1;
  FaddNoSwap:

  # A1. Unpack
  $eu = $eu >>u 53;
  $ev = $ev >>u 53;
  $tmp = #FFF0000000000000;
  $fu = $u andn $tmp;
  $fv = $v andn $tmp;
  $fu = $fu | #0010000000000000;
  $fv = $fv | #0010000000000000;
  $su = $u >>u 63;
  $sv = $v >>u 63;

  # A3. Set ew = eu
  $e = $eu;
  $s = $su;

  # A4. Test eu - ev
  $d = $eu - $ev;
  $tmp = $d - 55;   # 53 + 2
  goto FaddScale if $tmp is neg;
  # eu - ev >= p+2: result is just fu
  $f = $fu;
  $f0 = 0;
  goto FaddNormalize;

  FaddScale:
  # A5. Scale right: shift (fv, 0) right d places
  $tmp = - $d;
  $tmp = $tmp + 64;   # 64 - d
  $f0 = $fv <<u $tmp;   # low bits of shifted fv
  $fv = $fv >>u $d;

  # A6. Add or subtract based on signs
  $tmp = $su - $sv;
  goto FaddSubtract if $tmp is nonzero;
  # Same sign: add
  $f = $fu +u $fv;
  goto FaddNormalize;

  FaddSubtract:
  # Different signs: subtract
  $f0 = -u $f0;
  $carry = 0;
  $carry ?= 1 if $f0 is nonzero;
  $f = $fu -u $fv;
  $f = $f -u $carry;
  $tmp = $f | $f0;
  goto FaddError if $tmp is zero;   # result is exactly zero

  FaddNormalize:
  # N1. Test f
  $tmp = $f >>u 53;
  goto FaddN4 if $tmp is pos;
  $tmp = $f | $f0;
  goto FaddError if $tmp is zero;   # underflow

  FaddN2:
  $tmp = $f >>u 52;
  goto FaddN5 if $tmp is pos;

  # N3. Scale left
  $carry = 0;
  $carry ?= 1 if $f0 is neg;
  $f0 = $f0 <<u 1;
  $f = $f <<u 1;
  $f = $f +u $carry;
  $e = $e - 1;
  goto FaddN2;

  FaddN4:
  # N4. Scale right
  $carry = $f & 1;
```

```
  $carry = $carry <<u 63;
  $f0 = $f0 >>u 1;
  $f0 = $f0 +u $carry;
  $f = $f >>u 1;
  $e = $e + 1;

  FaddN5:
  # N5. Round
  $tmp = #8000000000000000;
  $tmp = $f0 cmpu $tmp;
  $carry = 1 if $f is odd;   # CSOD: round up if f odd and f0 >= 1/2
  $carry = $tmp if $f is even;   # CSEV: round up if f even and f0 > 1/2
  $carry ?= $carry if $tmp is nonneg;   # ZSNN: zero carry if f0 < 1/2
  $f = $f +u $carry;
  $f0 = 0;
  $tmp = $f >>u 53;
  goto FaddN4 if $tmp is pos;   # rounding overflow

  # N6. Check e
  $tmp = #7FE;
  $tmp = $tmp - $e;
  goto FaddError if $tmp is neg;   # overflow
  goto FaddError if $e is nonpos;   # underflow

  # N7. Pack
  $w = $s <<u 63;
  $tmp = $e <<u 52;
  $w = $w | $tmp;
  $f andnh= #FFF0;
  $w = $w | $f;
  return $w;

  FaddError:
  trap 0, 1, 0;
  $w = 0;
  return $w;
end

segment code;
register $u, $v;
register $w = $1;
register $tmp = $4;

# test: -100.0 + 103.0
$tmp = 100;
$tmp = - $tmp;
$u = flot. $tmp;
$tmp = 103;
$v = flot. $tmp;
$w = call Fadd($u, $v);

trap 0, HALT, 0;
```

progm: Programs M and D – Floating-point multiplication and division

progm.mms

```
<<<---
% Author M.Ruckert
 LOC #100
w IS $0 return value
u IS $0 1. parameter
v IS $1 2. parameter

s IS $2 $w$ unpacked
e IS $3
f IS $4
f0 IS $5 The low 64 bit of $f$
carry IS $13 The shuttle between $f$ and $f_0$.

eu IS $7 $u$ unpacked
fu IS $8

ev IS $10 $v$ unpacked
fv IS $11

d IS $12 $e_u - e_v$
tmp IS $14

--->>>
Fmul SLU eu,u,1; SLU ev,v,1 Remove sign bit
 SRU eu,eu,53; SRU ev,ev,53 M1. Unpack.
       SETH tmp,#FFF0   Get sign and exponent mask.
 ANDN fu,u,tmp; ANDN fv,v,tmp  Remove sign and exponet bits.
 INCH fu,#0010; INCH fv,#0010  Add hidden bit.
 XOR s,u,v; SRU s,s,63  $s_w is s_u\cdot s_v$

 SLU fv,fv,6; SLU fu,fu,6     M2. Operate.
 MULU f0,fu,fv; GET f,rH  $(f,f_0) \is 2^{52+6}f_u2^{52+6}f_v=2^{52+64}f_uf_v$
 ADD e,eu,ev; SET tmp,1023; SUB e,e,tmp $e\is e_u+e_v-q$
 JMP Normalize M3. Normalize.

Fdiv SLU eu,u,1; SLU ev,v,1  Remove sign bit
 SRU eu,eu,53; SRU ev,ev,53 M1. Unpack.
       SETH tmp,#FFF0   Get sign and exponent mask.
 ANDN fu,u,tmp; ANDN fv,v,tmp  Remove sign and exponent bits.
 INCH fu,#0010; INCH fv,#0010  Add hidden bit.
 XOR s,u,v; SRU s,s,63  $s_w is s_u\cdot s_v$


 SLU fv,fv,11   M2. Operate. $f_v is 2^{11}f_v$

 PUT rD,fu; SET tmp,0;  DIVU f,tmp,fv $(f,f_0)\is 2^{52+64}f_u/(2^{52+11}f_v)=2^{53}f_u/f_v$
 GET  tmp,rR; PUT rD,tmp
 SET tmp,0; DIVU f0,tmp,fv
 SUB e,eu,ev; INCL e,1023-1 $e\is e_u-e_v+q-1$

 JMP Normalize M3. Normalize.
<<<---


Normalize  SRU tmp,f,53; BP tmp,4F N1. Test $f$.
    OR tmp,f,f0; BZ tmp,Error underflow

2H    SRU tmp,f,52; BP tmp,5F N2. Is $f$ normalized?

    ZSN carry,f0,1; SLU f0,f0,1 N3. Scale left.
    SLU f,f,1; ADDU f,f,carry
    SUB e,e,1
    JMP 2B

4H    AND  carry,f,1 N4. Scale right.
    SLU carry,carry,63
    SRU f0,f0,1; ADDU f0,f0,carry
    SRU f,f,1
    ADD e,e,1

5H    SETH tmp,#8000 N5. Round.
    CMPU tmp,f0,tmp Compare $f_0$ to ${1\over 2}$
    CSOD carry,f,1   $f$ is odd. Round up if $f_0 \ge {1\over 2}$
    CSEV carry,f,tmp $f$ is even. Round up if $f_0 > {1\over 2}$
    ZSNN carry,tmp,carry Round down if $f_0 < {1\over 2}$
    ADDU f,f,carry
    SET f0,0
    SRU tmp,f,53; BP tmp,4B  Rounding overflow

    SET tmp,#7FE; CMP tmp,e,tmp    N6. Check $e$.
    BP tmp,Error Overflow
    BNP e,Error  Underflow

    SLU w,s,63  N5. Pack.
    SLU tmp,e,52; OR w,w,tmp
    ANDNH f,#FFF0 remove hidden bit
    OR w,w,f

    SET $0,w
    POP 1,0

Main    FLOT    $2,2
    FLOT    $3,5
    PUSHJ   $1,Fmul
    FLOT    $2,30
    PUSHJ   $0,Fdiv
    SET    $255,0
    TRAP    0,Halt,0

Error    SET    $255,1
    TRAP    0,Halt,0
--->>>
```

progm.plm

```
# Software floating-point multiplication and division (TAOCP 4.2....
# Fmul($0, $1): w = u * v (IEEE 754 double, software)
# Fdiv($0, $1): w = u / v (IEEE 754 double, software)
# Shared normalization routine.

# Normalize a (f, f0) significand with exponent e and sign s.
# Inputs: f=$4, f0=$5, e=$3, s=$2; carry=$13; tmp=$14
# Returns: $0 = packed double
# Based on an .mms file by M.Ruckert
def FmulNorm($w, $1):
  # This is a shared stub - actual procedures inline the normalizat...
  register $s, $e, $f, $f0;
  register $carry, $tmp;

  # N1. Test f
  Norm1:
  $tmp = $f >>u 53;
  goto Norm4 if $tmp is pos;
  $tmp = $f | $f0;
  goto NormError if $tmp is zero;   # underflow

  Norm2:
  $tmp = $f >>u 52;
  goto Norm5 if $tmp is pos;

  # N3. Scale left
  $carry = 0;
  $carry ?= 1 if $f0 is neg;
  $f0 = $f0 <<u 1;
  $f = $f <<u 1;
  $f = $f +u $carry;
  $e = $e - 1;
  goto Norm2;

  Norm4:
  # N4. Scale right
  $carry = $f & 1;
  $carry = $carry <<u 63;
  $f0 = $f0 >>u 1;
  $f0 = $f0 +u $carry;
  $f = $f >>u 1;
  $e = $e + 1;

  Norm5:
  # N5. Round
  $tmp = #8000000000000000;
  $tmp = $f0 cmpu $tmp;
  $carry = 1 if $f is odd;
  $carry = $tmp if $f is even;
  $carry ?= $carry if $tmp is nonneg;
  $f = $f +u $carry;
  $f0 = 0;
  $tmp = $f >>u 53;
  goto Norm4 if $tmp is pos;

  # N6. Check e
  $tmp = #7FE;
  $tmp = $tmp - $e;
  goto NormError if $tmp is neg;
  goto NormError if $e is nonpos;

  # N7. Pack
  $w = $s <<u 63;
  $tmp = $e <<u 52;
  $w = $w | $tmp;
  $f andnh= #FFF0;
  $w = $w | $f;
  return $w;

  NormError:
  trap 0, 1, 0;
  $w = 0;
  return $w;
end

def Fmul($w, $v):
  register $u = $0;
  register $s, $e, $f, $f0;
  register $carry = $13;
  register $eu, $fu;
  register $ev, $fv;
  register $tmp = $14;

  # M1. Unpack
  $eu = $u <<u 1;
  $ev = $v <<u 1;
  $eu = $eu >>u 53;
  $ev = $ev >>u 53;
  $tmp = #FFF0000000000000;
  $fu = $u andn $tmp;
  $fv = $v andn $tmp;
  $fu = $fu | #0010000000000000;
  $fv = $fv | #0010000000000000;
  $s = $u ^ $v;
  $s = $s >>u 63;   # XOR signs

  # M2. Multiply: (f,f0) = fu * fv scaled
  $fu = $fu <<u 6;
  $fv = $fv <<u 6;
  $f0 = $fu *$u $fv;
  $f = $rH;   # high 64 bits: (f,f0) = 2^(52+6) * fu * 2^(52+6) * fv
  $e = $eu + $ev;
  $e = $e - 1023;   # e = eu + ev - q

  # Normalize using shared code (inline)
  Fmul_Norm1:
  $tmp = $f >>u 53;
  goto Fmul_Norm4 if $tmp is pos;
  $tmp = $f | $f0;
  goto FmulError if $tmp is zero;
```

```
  Fmul_Norm2:
  $tmp = $f >>u 52;
  goto Fmul_Norm5 if $tmp is pos;
  $carry = 0;
  $carry ?= 1 if $f0 is neg;
  $f0 = $f0 <<u 1;
  $f = $f <<u 1;
  $f = $f +u $carry;
  $e = $e - 1;
  goto Fmul_Norm2;

  Fmul_Norm4:
  $carry = $f & 1;
  $carry = $carry <<u 63;
  $f0 = $f0 >>u 1;
  $f0 = $f0 +u $carry;
  $f = $f >>u 1;
  $e = $e + 1;

  Fmul_Norm5:
  $tmp = #8000000000000000;
  $tmp = $f0 cmpu $tmp;
  $carry = 1 if $f is odd;
  $carry = $tmp if $f is even;
  $carry ?= $carry if $tmp is nonneg;
  $f = $f +u $carry;
  $f0 = 0;
  $tmp = $f >>u 53;
  goto Fmul_Norm4 if $tmp is pos;
  $tmp = #7FE;
  $tmp = $tmp - $e;
  goto FmulError if $tmp is neg;
  goto FmulError if $e is nonpos;
  $w = $s <<u 63;
  $tmp = $e <<u 52;
  $w = $w | $tmp;
  $f andnh= #FFF0;
  $w = $w | $f;
  return $w;

  FmulError:
  trap 0, 1, 0;
  $w = 0;
  return $w;
end

def Fdiv($w, $v):
  register $u = $0;
  register $s, $e, $f, $f0;
  register $carry = $13;
  register $eu, $fu;
  register $ev, $fv, $d;
  register $tmp = $14;

  # M1. Unpack
  $eu = $u <<u 1;
  $ev = $v <<u 1;
  $eu = $eu >>u 53;
  $ev = $ev >>u 53;
  $tmp = #FFF0000000000000;
  $fu = $u andn $tmp;
  $fv = $v andn $tmp;
  $fu = $fu | #0010000000000000;
  $fv = $fv | #0010000000000000;
  $s = $u ^ $v;
  $s = $s >>u 63;

  # M2. Divide: (f,f0) = fu / fv scaled
  $fv = $fv <<u 11;   # fv = 2^11 * fv
  $rD = $fu;
  $d = 0;
  $f = $d /$u $fv;   # f = (fu * 2^64 + 0) / (2^11 * fv) = 2^53 * fu/fv
  $tmp = $rR;
  $rD = $tmp;
  $d = 0;
  $f0 = $d /$u $fv;   # f0 = remainder / fv
  $e = $eu - $ev;
  $e = $e + 1022;   # e = eu - ev + q - 1

  # Normalize
  Fdiv_Norm1:
  $tmp = $f >>u 53;
  goto Fdiv_Norm4 if $tmp is pos;
  $tmp = $f | $f0;
  goto FdivError if $tmp is zero;

  Fdiv_Norm2:
  $tmp = $f >>u 52;
  goto Fdiv_Norm5 if $tmp is pos;
  $carry = 0;
  $carry ?= 1 if $f0 is neg;
  $f0 = $f0 <<u 1;
  $f = $f <<u 1;
  $f = $f +u $carry;
  $e = $e - 1;
  goto Fdiv_Norm2;

  Fdiv_Norm4:
  $carry = $f & 1;
  $carry = $carry <<u 63;
  $f0 = $f0 >>u 1;
  $f0 = $f0 +u $carry;
  $f = $f >>u 1;
  $e = $e + 1;

  Fdiv_Norm5:
  $tmp = #8000000000000000;
  $tmp = $f0 cmpu $tmp;
  $carry = 1 if $f is odd;
  $carry = $tmp if $f is even;
  $carry ?= $carry if $tmp is nonneg;
  $f = $f +u $carry;
  $f0 = 0;
  $tmp = $f >>u 53;
  goto Fdiv_Norm4 if $tmp is pos;
```

```
  $tmp = #7FE;
  $tmp = $tmp - $e;
  goto FdivError if $tmp is neg;
  goto FdivError if $e is nonpos;
  $w = $s <<u 63;
  $tmp = $e <<u 52;
  $w = $w | $tmp;
  $f andnh= #FFF0;
  $w = $w | $f;
  return $w;

  FdivError:
  trap 0, 1, 0;
  $w = 0;
  return $w;
end

segment code;
register $u, $v;
register $w = $1;
register $tmp = $4;

# test: 2.0 * 5.0 = 10.0
$tmp = 2;
$u = flot. $tmp;
$tmp = 5;
$v = flot. $tmp;
$w = call Fmul($u, $v);

# test: 10.0 / 30.0
$tmp = 30;
$v = flot. $tmp;
$w = call Fdiv($w, $v);

trap 0, HALT, 0;
```

TAOCP 4.2.3 – Double-Length Arithmetic

ex6: Double-precision example (exercise 6)

ex6.mms

```
<<<---
% Author M.Ruckert
  LOC #100

Main  FLOT $2,15
  PUSHJ $1,ToDouble

  PUSHJ $0,ToSingle

  TRAP 0,Halt,0

--->>>
    PREFIX :ToDouble:
x    IS $0 Parameter
x0    IS $2
s    IS $3 Local variables
e    IS $4
:ToDouble    SRU s,x,63   Unpack.
     SLU e,x,1; SRU e,x,53
     ANDNH x,#FFF0   Remove sign and exponent bits.
     INCL e,#3FFF-#3FF  Convert.
     SLU x0,x,64-(52-48)
     SRU x,x,(52-48)
     SLU s,s,63   Pack.
     SLU e,e,48;
    OR  $1,x,s; OR $1,$1,e Return $(x_1,x_0)$.
    SET $0,x0
     POP 2,0

    PREFIX :ToSingle:
x1  IS $0 Parameter
x0  IS $1
s  IS $2 Local variables
e  IS $3
f  IS $4
carry  IS $5
tmp  IS $6
:ToSingle SLU e,x1,1; SRU e,e,49 Unpack.
         SRU s,x1,63
  ANDNH x1,#FFFF
  SET tmp,#3FFF-#3FF  Convert.
  SUB e,e,tmp
  BNP e,Error   Underflow.
  SET tmp,#7FF
  CMP tmp,e,tmp
  BNN tmp,Error  Overflow.
  SLU f,x1,(52-48)
  SRU tmp,x0,64-(52-48)
  OR f,f,tmp
  SLU x0,x0,52-48
  SETH tmp,#8000  Round. see 4.2.1 Program A
     CMPU tmp,x0,tmp
     CSOD carry,f,1
     CSEV carry,f,tmp
     ZSNN carry,tmp,carry
     ADDU f,f,carry
     SLU $0,s,63   Pack.
     SLU e,e,52; OR $0,$0,e
     OR $0,$0,f
     POP 1,0
<<<---
Error  SET $255,1
  TRAP 0,:Halt,0
```

ex6.plm

```
# Convert between 64-bit and 80-bit floating-point formats (TAOCP...

# Based on an .mms file by M.Ruckert
def ToDouble($x) -> $lo, $hi:
  register $s = $3;
  register $e = $4;
  const BIAS_DELTA = 15360;
  const SHIFT = 4;

  $s = $x >>u 63;
  $e = $x <<u 1;
  $e = $x >>u 53;
  $x andnh= #FFF0;
  $e = $e + BIAS_DELTA;
  $lo = $x <<u 60;
  $hi = $x >>u SHIFT;
  $s = $s <<u 63;
  $e = $e <<u 48;
  $hi = $hi | $s;
  $hi = $hi | $e;
  return $lo, $hi;
end

def ToSingle($x1, $x0):
  register $s = $2;
  register $e = $3;
  register $f = $4;
  register $carry = $5;
  register $tmp = $6;
  const SHIFT = 4;

  $e = $x1 <<u 1;
  $e = $e >>u 49;
  $s = $x1 >>u 63;
  $x1 andnh= #FFFF;
  $tmp = #3C00;
  $e = $e - $tmp;
  goto Error if $e is nonpos;
  $tmp = #7FF;
  $tmp = $e cmpu $tmp;
  goto Error if $tmp is nonneg;
  $f = $x1 <<u SHIFT;
  $tmp = $x0 >>u 60;
  $f = $f | $tmp;
  $x0 = $x0 <<u SHIFT;
  $tmp = #8000000000000000;
  $tmp = $x0 cmpu $tmp;
  $carry = 1 if $f is odd;
  $carry = $tmp if $f is even;
  $carry ?= $carry if $tmp is nonneg;
  $f = $f +u $carry;
  $x1 = $s <<u 63;
  $e = $e <<u 52;
  $x1 = $x1 | $e;
  $x1 = $x1 | $f;
  return $x1;

  Error:
  $255 = 1;
  trap 0, HALT, 0;
end

segment code;
register $x1 = $0;
register $x0 = $1;
register $tmp = $2;

$tmp = 15;
$x1 = flot. $tmp;
$x0, $x1 = call ToDouble($x1);
$x1 = call ToSingle($x1, $x0);

trap 0, HALT, 0;
```

progda: Program DA – Double-precision addition and subtraction

progda.mms

```
<<<---
% Author M.Ruckert
 LOC #100

u1 IS $0 1. parameter
u0 IS $1 1. parameter
v1 IS $2 2. parameter
v0 IS $3 2. parameter
w1 IS $4 return value
w0 IS $5 return value

s IS $6 $w$ unpacked
e IS $7
carry IS $8 The shuttle between $x_1$ and $x_0$.

su IS $9 $u$ unpacked
eu IS $10

sv IS $11 $v$ unpacked
ev IS $12

d IS $13 $e_u - e_v$
tmp IS $14

--->>>
DFsub SETH tmp,#8000; XOR v1,v1,tmp Change sign of operand.
DFadd SLU eu,u1,1; SLU ev,v1,1  Remove sign bit.
 CMPU tmp,eu,ev   A2. Assume $e_u \ge e_v$.
 BP tmp,OF          Branch if $ (e_u, u_1, u_0) \ge(e_v, v_1,v_0) $
  BN tmp,1F; CMPU tmp,u0,v0; BNN tmp,OF
1H SET tmp,u1; SET u1,v1; SET v1,tmp       else swap $u$ and $v$
 SET tmp,u0; SET u0,v0; SET v0,tmp
 SLU eu,u1,1; SLU ev,v1,1         and remove sign bit again.

0H SRU eu,eu,49; SRU ev,ev,49 A1. Unpack.
 SRU su,u1,63; SRU sv,v1,63 Get sign bit.
 ANDNH u1,#FFFF; ANDNH v1,#FFFF  Remove sign and exponent bits.
 INCH u1,#0001; INCH v1,#0001  Add hidden bit.

 SET e,eu; SET s,su  A3. Set $e_w is e_u$.

 SUB d,eu,ev  A4. Test $e_u-e_v$.
 CMP tmp,d,113+2; BN tmp,5F  $e_u-e_v \ge p+2$ ?
 SET w1,u1; SET w0,u0     $w \is u$
 JMP 7F

5H     CMP tmp,d,64; BN tmp,0F  A5. Scale right.
 SET v0,v1; SET  v1,0; SUB d,d,64    Truncating $v_0$ if $d\ge64$
0H SRU v0,v0,d
 NEG tmp,64,d; SLU carry,v1,tmp
 OR v0,v0,carry; SRU v1,v1,d Shift $(v_1,v_0)$ to the right $e_u-e_v$ places

 CMP tmp,su,sv; BNZ tmp,0F  Signs $s_u$ and $s_v$ differ.
 ADDU w0,u0,v0   A6. Add.
 CMP tmp,w0,u0; ZSN carry,tmp,1
 ADDU w1,u1,v1; ADDU w1,w1,carry
 JMP 7F

0H SUBU w0,u0,v0  A6. Subtract.
 CMP tmp,w0,u0; ZSP carry,tmp,1
 SUBU w1,u1,v1; SUBU w1,w1,carry
 OR tmp,w1,w0; BZ tmp,Error  Result is zero.
7H IS @      A7. Normalize.
DNormalize  SRU tmp,w1,49; BP tmp,4F N1. Test $f$. If $w\ge 2$, scale right.
    OR tmp,w1,w0; BZ tmp,Error underflow

2H    SRU tmp,w1,48; BP tmp,5F N2. Is $w$ normalized?

    ZSN carry,w0,1; SLU w0,w0,1 N3. Scale left.
    SLU w1,w1,1; ADDU w1,w1,carry
    SUB e,e,1
    JMP 2B

4H    AND  carry,w1,1 N4. Scale right.
    SLU carry,carry,63
    SRU w0,w0,1; ADDU w0,w0,carry
    SRU w1,w1,1
    ADD e,e,1
     % N5. Round. omitted
5H    SET tmp,#7FFE; CMP tmp,e,tmp    N6. Check $e$.
    BP tmp,Error Overflow
    BNP e,Error  Underflow

    SLU s,s,63  N7. Pack.
    SLU e,e,48;
    ANDNH w1,#FFFF remove hidden bit
    OR   w1,w1,s; OR w1,w1,e
<<<---
    SET $0,w0
    SET $1,w1
    POP 2,0

Main    SETML   $2,0  1.0
    INCH    $2,#3FFF+10 exponent 10
    ORH    $2,#8000 negativ
    SET    $3,0  $2,$3 <- -1024.0
    SETML   $4,#0040  1.0000 0000 0100 0000
    INCH    $4,#3FFF+10 exponent 10
    ORH    $4,#0000 positive
    SET    $5,0  $4,$5 <- +1025.0
    PUSHJ   $1,DFadd

    SET    $255,0
    TRAP    0,Halt,0

Error    SET    $255,1
    TRAP    0,Halt,0
--->>>
```

progda.plm

```
# Double-double precision floating-point addition (TAOCP 4.2.3, A...
# DFsub(u1,u0,v1,v0): w = u - v (negate v1 sign then call DFadd)
# DFadd(u1,u0,v1,v0): w = u + v (80-bit extended precision)
# Parameters: $0=u1(high), $1=u0(low), $2=v1(high), $3=v0(low)
# Returns: $0=w1(high), $1=w0(low) [2 return values]

# Based on an .mms file by M.Ruckert
def DFsub($u1, $u0, $v1, $v0) -> $w1, $w0:
  register $tmp = $14;
  $tmp = #8000000000000000;
  $v1 = $v1 ^ $tmp;   # negate v by flipping sign bit
  goto DFaddBody;
end

def DFadd($u1, $u0, $v1, $v0) -> $w1, $w0:
  DFaddBody:
  register $s, $e, $carry, $su, $eu, $sv, $ev, $d, $tmp;

  # A2. Assume eu >= ev; compare and swap if needed
  $eu = $u1 <<u 1;
  $ev = $v1 <<u 1;
  $tmp = $eu cmpu $ev;
  goto DFadd_NoSwap if $tmp is pos;
  goto DFadd_Swap if $tmp is neg;
  # equal high words: compare low words
  $tmp = $u0 cmpu $v0;
  goto DFadd_NoSwap if $tmp is nonneg;
  DFadd_Swap:
  $tmp = $u1; $u1 = $v1; $v1 = $tmp;
  $tmp = $u0; $u0 = $v0; $v0 = $tmp;
  $eu = $u1 <<u 1;
  $ev = $v1 <<u 1;
  DFadd_NoSwap:

  # A1. Unpack
  $eu = $eu >>u 49;   # 15-bit exponent
  $ev = $ev >>u 49;
  $su = $u1 >>u 63;
  $sv = $v1 >>u 63;
  $u1 andnh= #FFFF;   # remove sign and exponent
  $v1 andnh= #FFFF;
  $u1 = $u1 | #0001000000000000;   # add hidden bit at position 48
  $v1 = $v1 | #0001000000000000;

  # A3. Set e = eu
  $e = $eu;
  $s = $su;

  # A4. Test d = eu - ev
  $d = $eu - $ev;
  $tmp = $d - 115;   # 113 + 2
  goto DFadd_Scale if $tmp is neg;
  # eu-ev >= p+2: result is just u
  $w1 = $u1;
  $w0 = $u0;
  goto DFadd_Normalize;

  DFadd_Scale:
  # A5. Scale right v by d places
  $tmp = $d - 64;
  goto DFadd_Scale2 if $tmp is neg;
  $v0 = $v1;
  $v1 = 0;
  $d = $d - 64;
  DFadd_Scale2:
  $v0 = $v0 >>u $d;
  $tmp = - $d;
  $tmp = $tmp + 64;
  $carry = $v1 <<u $tmp;
  $v0 = $v0 | $carry;
  $v1 = $v1 >>u $d;

  # A6. Add or subtract
  $tmp = $su - $sv;
  goto DFadd_Sub if $tmp is nonzero;
  # Add
  $w0 = $u0 +u $v0;
  $tmp = $w0 cmpu $u0;
  $carry = 0;
  $carry ?= 1 if $tmp is neg;
  $w1 = $u1 +u $v1;
  $w1 = $w1 +u $carry;
  goto DFadd_Normalize;

  DFadd_Sub:
  # Subtract
  $w0 = $u0 -u $v0;
  $tmp = $w0 cmpu $u0;
  $carry = 0;
  $carry ?= 1 if $tmp is pos;
  $w1 = $u1 -u $v1;
  $w1 = $w1 -u $carry;
  $tmp = $w1 | $w0;
  goto DFaddError if $tmp is zero;

  DFadd_Normalize:
  # N1. Test w1
  DFnorm1:
  $tmp = $w1 >>u 49;
  goto DFnorm4 if $tmp is pos;
  $tmp = $w1 | $w0;
  goto DFaddError if $tmp is zero;

  DFnorm2:
  $tmp = $w1 >>u 48;
  goto DFnorm5 if $tmp is pos;

  # N3. Scale left
  $carry = 0;
  $carry ?= 1 if $w0 is neg;
```

```
  $w0 = $w0 <<u 1;
  $w1 = $w1 <<u 1;
  $w1 = $w1 +u $carry;
  $e = $e - 1;
  goto DFnorm2;

  DFnorm4:
  # N4. Scale right
  $carry = $w1 & 1;
  $carry = $carry <<u 63;
  $w0 = $w0 >>u 1;
  $w0 = $w0 +u $carry;
  $w1 = $w1 >>u 1;
  $e = $e + 1;

  DFnorm5:
  # N6. Check e (no rounding in this version)
  $tmp = #7FFE;
  $tmp = $tmp - $e;
  goto DFaddError if $tmp is neg;
  goto DFaddError if $e is nonpos;

  # N7. Pack
  $tmp = $s <<u 63;
  $e = $e <<u 48;
  $w1 andnh= #FFFF;   # clear old sign+exp
  $w1 = $w1 | $tmp;
  $w1 = $w1 | $e;
  return $w1, $w0;

  DFaddError:
  trap 0, 1, 0;
  $w1 = 0;
  $w0 = 0;
  return $w1, $w0;
end

segment code;
register $u1, $u0, $v1, $v0;
register $w1, $w0;

# test: -1024.0 + 1025.0 = 1.0 using 80-bit format
# Load -1024.0 in 80-bit: sign=1, exp=16383+10=16393=#4009, manti...
$u1 = #C00A800000000000;   # sign=1, exp=16394, hidden+0
$u0 = 0;
# Load +1025.0: exp=16394, mantissa bit
$v1 = #400A802000000000;
$v0 = 0;
$w1, $w0 = call DFadd($u1, $u0, $v1, $v0);

trap 0, HALT, 0;
```

progdd: Program DD – Double-precision division

progdd.mms

```
<<<---
% Author M.Ruckert
 LOC #100
u1 IS $0 1. parameter
u0 IS $1 1. parameter
v1 IS $2 2. parameter
v0 IS $3 2. parameter
w1 IS $4 return value
w0 IS $5 return value

s IS $6 $w$ unpacked
e IS $7
carry IS $8 The shuttle between $x_1$ and $x_0$.

su IS $9 $u$ unpacked
eu IS $10

sv IS $11 $v$ unpacked
ev IS $12

d IS $13 $e_u - e_v$
tmp IS $14

--->>>
DFdiv SLU eu,u1,1; SLU ev,v1,1   M1. Unpack.
 SRU eu,eu,49; SRU ev,ev,49
 XOR s,u1,v1; SRU s,s,63  $s_w is s_u\cdot s_v$
 ANDNH u1,#FFFF; ORH   u1,#0001
 SLU v1,v1,15; ORH  v1,#8000  $(v_1,v_0) is (v_1,v_0)2^{15}$
       SRU carry,v0,64-15; ADDU v1,v1,carry
 SLU v0,v0,15
      % M2. Operate.
 PUT rD,u1;  DIVU w1,u0,v1       $(u_12^{64}+ u_0)/(v_12^{15})$
 PUT rD,0; GET tmp,rR; DIVU w0,tmp,v1  $(w_1,w_0)\is 2(u_1,u_0)/v_1$
 SUB e,eu,ev; INCL e,#3FFF-1    $e\is e_u-e_v+q-1$

<<<--- BZ v0,DNormalize    not sure if that would be an optimization --->>>
 MULU d,w1,v0     $d \is w_1v_0$
 GET tmp,rH; PUT rD,tmp; DIVU d,d,v1        $d \is w_1v_0/v_1$
 CMPU tmp,w0,d; ZSN carry,tmp,1  $\.{carry}\is 1$ if $w_0<d$
 SUBU w0,w0,d; SUBU w1,w1,carry

 JMP DNormalize    M3. Normalize.
<<<---
DNormalize  SRU tmp,w1,49; BP tmp,4F N1. Test $f$. If $w\ge 2$, scale right.
    OR tmp,w1,w0; BZ tmp,Error underflow

2H    SRU tmp,w1,48; BP tmp,5F N2. Is $w$ normalized?

    ZSN carry,w0,1; SLU w0,w0,1 N3. Scale left.
    SLU w1,w1,1; ADDU w1,w1,carry
    SUB e,e,1
    JMP 2B

4H    AND  carry,w1,1 N4. Scale right.
    SLU carry,carry,63
    SRU w0,w0,1; ADDU w0,w0,carry
    SRU w1,w1,1
    ADD e,e,1
     % N5. Round. omitted
5H    SET tmp,#7FFE; CMP tmp,e,tmp    N6. Check $e$.
    BP tmp,Error Overflow
    BNP e,Error  Underflow

    SLU s,s,63  N7. Pack.
    SLU e,e,48;
    ANDNH w1,#FFFF remove hidden bit
    OR   w1,w1,s; OR w1,w1,e

    SET $0,w0
    SET $1,w1
    POP 2,0
Main    SETML   $2,0  1.0
    INCH    $2,#3FFF+10 exponent 10
    ORH    $2,#8000 negativ
    SET    $3,0  $2,$3 <- -1024.0
    SETML   $4,#0040  1.0000 0000 0100 0000
    INCH    $4,#3FFF+10 exponent 10
    ORH    $4,#0000 positive
    SET    $5,0  $4,$5 <- +1025.0
    PUSHJ   $1,DFdiv

    SET    $255,0
    TRAP    0,Halt,0

Error    SET    $255,1
    TRAP    0,Halt,0
--->>>
```

progdd.plm

```
# Double-double precision floating-point division (TAOCP 4.2.3, A...
# DFdiv(u1,u0,v1,v0): w = u / v (80-bit extended precision)
# Parameters: $0=u1(high), $1=u0(low), $2=v1(high), $3=v0(low)
# Returns: $0=w1(high), $1=w0(low)

# Based on an .mms file by M.Ruckert
def DFdiv($u1, $u0, $v1, $v0) -> $w1, $w0:
  register $s, $e, $carry;
  register $eu = $10;
  register $fu = $0;# alias u1 after unpacking
  register $ev = $12;
  register $fv = $2;# alias v1 after unpacking
  register $d, $tmp;

  # M1. Unpack
  $eu = $u1 <<u 1;
  $ev = $v1 <<u 1;
  $eu = $eu >>u 49;
  $ev = $ev >>u 49;
  $s = $u1 ^ $v1;
  $s = $s >>u 63;   # sign of result
  $u1 andnh= #FFFF;   # fu = u1 without sign/exp
  $v1 andnh= #FFFF;
  $u1 = $u1 | #0001000000000000;   # add hidden bit
  $v1 = $v1 | #0001000000000000;

  # M2. Divide using DIVU
  # Shift v left 15 bits: fv = v1 * 2^15, incorporate v0
  $carry = $v0 >>u 49;
  $v1 = $v1 <<u 15;
  $v1 = $v1 | $carry;
  $v0 = $v0 <<u 15;

  $rD = $u1;
  $d = 0;
  $w1 = $d /u $v1;   # w1 = (u1*2^64 + u0) / (v1*2^15)
  $tmp = 0;
  $rD = $tmp;
  $tmp = $rR;
  $rD = $tmp;
  $d = 0;
  $w0 = $d /u $v1;   # w0 = remainder / v1

  $e = $eu - $ev;
  $e = $e + 16382;   # e = eu - ev + q - 1; q = #3FFF, q-1 = #3FFE = 16382

  # Correct w0 for v0 contribution: subtract w1*v0/v1
  $d = $w1 *u $v0;
  $tmp = $rH;
  $rD = $tmp;
  $d = $d /u $v1;
  $tmp = $w0 cmpu $d;
  $carry = 0;
  $carry ?= 1 if $tmp is neg;
  $w0 = $w0 -u $d;
  $w1 = $w1 -u $carry;

  # Normalize
  DFdiv_N1:
  $tmp = $w1 >>u 49;
  goto DFdiv_N4 if $tmp is pos;
  $tmp = $w1 | $w0;
  goto DFdivError if $tmp is zero;

  DFdiv_N2:
  $tmp = $w1 >>u 48;
  goto DFdiv_N5 if $tmp is pos;
  $carry = 0;
  $carry ?= 1 if $w0 is neg;
  $w0 = $w0 <<u 1;
  $w1 = $w1 <<u 1;
  $w1 = $w1 +u $carry;
  $e = $e - 1;
  goto DFdiv_N2;

  DFdiv_N4:
  $carry = $w1 & 1;
  $carry = $carry <<u 63;
  $w0 = $w0 >>u 1;
  $w0 = $w0 +u $carry;
  $w1 = $w1 >>u 1;
  $e = $e + 1;

  DFdiv_N5:
  # N6. Check e
  $tmp = #7FFE;
  $tmp = $tmp - $e;
  goto DFdivError if $tmp is neg;
  goto DFdivError if $e is nonpos;

  # N7. Pack
  $tmp = $s <<u 63;
  $e = $e <<u 48;
  $w1 andnh= #FFFF;
  $w1 = $w1 | $tmp;
  $w1 = $w1 | $e;
  return $w1, $w0;

  DFdivError:
  trap 0, 1, 0;
  $w1 = 0;
  $w0 = 0;
  return $w1, $w0;
end

segment code;
register $u1, $u0, $v1, $v0;
register $w1, $w0;

# test: -1024.0 / 1025.0
```

```
$u1 = #C00A800000000000;
$u0 = 0;
$v1 = #400A802000000000;
$v0 = 0;
$w1, $w0 = call DFdiv($u1, $u0, $v1, $v0);

trap 0, HALT, 0;
```

progdmsimple: Simplified double-precision multiplication

progdmsimple.mms

```
<<<---
% Author M.Ruckert
 LOC #100
u1 IS $0 1. parameter
u0 IS $1 1. parameter
v1 IS $2 2. parameter
v0 IS $3 2. parameter
w1 IS $4 return value
w0 IS $5 return value
w00 IS $6

s IS $7 $w$ unpacked
e IS $8
carry IS $9 The shuttle between $x_1$ and $x_0$.

su IS $10 $u$ unpacked
eu IS $11

sv IS $12 $v$ unpacked
ev IS $13

d IS $14 $e_u - e_v$
tmp IS $15

--->>>
DFmul XOR s,u1,v1; SRU s,s,63  $s_w is s_u\cdot s_v$
 SLU eu,u1,1; SLU ev,v1,1  M1. Unpack.
 SRU eu,eu,49; SRU ev,ev,49
 ANDNH u1,#FFFF; ORH   u1,#0001
 ANDNH v1,#FFFF; ORH   v1,#0001
       % M2. Operate.
 MULU tmp,u1,v0; GET w0,rH    $(w_0,\cdot) \is 2^{48+64}u_12^{64}v_0$
 MULU tmp,u0,v1;
 GET     tmp,rH; ADDU w0,w0,tmp   $w_0 \is w_0 + 2^{48}u_0v_1$
 MULU tmp,u1,v1; GET w1,rH   $(w_1,\cdot) \is 2^{48}u_12^{48}v_1=2^{32+64}u_1v_1$
 ADDU w0,w0,tmp          $w_0 \is w_0 +\cdot$
 CMPU tmp,w0,tmp; ZSN carry,tmp,1  $\.{carry} \is 1$ if $w_0+\cdot<\cdot$
 ADDU w1,w1,carry
 ADD e,eu,ev
 SLU w1,w1,16
 SRU carry,w0,64-16; ADDU w1,w1,carry
 SLU w0,w0,16
 SET  tmp,#3FFF; SUB e,e,tmp   $e\is e_u+e_v-q$

 JMP DNormalize        M3. Normalize.
<<<--- 28 instructions

DNormalize  SRU tmp,w1,49; BP tmp,4F N1. Test $f$. If $w\ge 2$, scale right.
    OR tmp,w1,w0; BZ tmp,Error underflow

2H    SRU tmp,w1,48; BP tmp,5F N2. Is $w$ normalized?

    ZSN carry,w0,1; SLU w0,w0,1 N3. Scale left.
    SLU w1,w1,1; ADDU w1,w1,carry
    SUB e,e,1
    JMP 2B

4H    AND  carry,w1,1 N4. Scale right.
    SLU carry,carry,63
    SRU w0,w0,1; ADDU w0,w0,carry
    SRU w1,w1,1
    ADD e,e,1
     % N5. Round. omitted
5H    SET tmp,#7FFE; CMP tmp,e,tmp    N6. Check $e$.
    BP tmp,Error Overflow
    BNP e,Error  Underflow

    SLU s,s,63  N7. Pack.
    SLU e,e,48;
    ANDNH w1,#FFFF remove hidden bit
    OR   w1,w1,s; OR w1,w1,e

    SET $0,w0
    SET $1,w1
    POP 2,0
Main    SETML   $2,0  1.0
    INCH    $2,#3FFF+10 exponent 10
    ORH    $2,#8000 negativ
    SET    $3,0  $2,$3 <- -1024.0
    SETML   $4,#0040  1.0000 0000 0100 0000
    INCH    $4,#3FFF+10 exponent 10
    ORH    $4,#0000 positive
    SET    $5,0  $4,$5 <- +1025.0
    PUSHJ   $1,DFdiv

    SET    $255,0
    TRAP    0,Halt,0

Error    SET    $255,1
    TRAP    0,Halt,0
--->>>
```

progdmsimple.plm

```
# Double-double precision floating-point multiplication (TAOCP 4....
# DFmul(u1,u0,v1,v0): w = u * v (80-bit extended precision)
# Parameters: $0=u1(high), $1=u0(low), $2=v1(high), $3=v0(low)
# Returns: $0=w1(high), $1=w0(low)

# Based on an .mms file by M.Ruckert
def DFmul($u1, $u0, $v1, $v0) -> $w1, $w0:
  register $s, $e, $carry;
  register $eu = $11;
  register $ev, $d, $tmp;

  # M1. Unpack
  $s = $u1 ^ $v1;
  $s = $s >>u 63;   # sign of result
  $eu = $u1 <<u 1;
  $ev = $v1 <<u 1;
  $eu = $eu >>u 49;
  $ev = $ev >>u 49;
  $u1 andnh= #FFFF;   # remove sign and exponent
  $v1 andnh= #FFFF;
  $u1 = $u1 | #0001000000000000;   # add hidden bit at position 48
  $v1 = $v1 | #0001000000000000;

  # M2. Multiply (u1,u0) * (v1,v0) using MULU and $rH
  # Compute (w1, w0) = (u1,u0) * (v1,v0) in 128-bit arithmetic
  # w0 contributions from cross terms
  $tmp = $u1 *u $v0;   # u1 * v0
  $w0 = $rH;   # high bits of u1*v0
  $tmp = $u0 *u $v1;
  $d = $rH;
  $w0 = $w0 +u $d;   # w0 += high(u0*v1)
  $tmp = $u0 *u $v1;
  $tmp = $w0 + $tmp;
  $carry = 0;
  $carry ?= 1 if $tmp is zero;
  # w1 contributions
  $tmp = $u1 *u $v1;
  $w1 = $rH;
  $w1 = $w1 +u $carry;

  $w0 = $w0 +u $tmp;
  $tmp = $w0 cmpu $tmp;
  $carry = 0;
  $carry ?= 1 if $tmp is neg;
  $w1 = $w1 +u $carry;

  $e = $eu + $ev;
  # Align: shift left 16 to bring hidden bit to position 48
  $w1 = $w1 <<u 16;
  $carry = $w0 >>u 48;
  $w1 = $w1 +u $carry;
  $w0 = $w0 <<u 16;
  $tmp = #3FFF;
  $e = $e - $tmp;   # unbias: e = eu + ev - q

  # Normalize
  DFmul_N1:
  $tmp = $w1 >>u 49;
  goto DFmul_N4 if $tmp is pos;
  $tmp = $w1 | $w0;
  goto DFmulError if $tmp is zero;

  DFmul_N2:
  $tmp = $w1 >>u 48;
  goto DFmul_N5 if $tmp is pos;
  $carry = 0;
  $carry ?= 1 if $w0 is neg;
  $w0 = $w0 <<u 1;
  $w1 = $w1 <<u 1;
  $w1 = $w1 +u $carry;
  $e = $e - 1;
  goto DFmul_N2;

  DFmul_N4:
  $carry = $w1 & 1;
  $carry = $carry <<u 63;
  $w0 = $w0 >>u 1;
  $w0 = $w0 +u $carry;
  $w1 = $w1 >>u 1;
  $e = $e + 1;

  DFmul_N5:
  # N6. Check e
  $tmp = #7FFE;
  $tmp = $tmp - $e;
  goto DFmulError if $tmp is neg;
  goto DFmulError if $e is nonpos;

  # N7. Pack
  $tmp = $s <<u 63;
  $e = $e <<u 48;
  $w1 andnh= #FFFF;
  $w1 = $w1 | $tmp;
  $w1 = $w1 | $e;
  return $w1, $w0;

  DFmulError:
  trap 0, 1, 0;
  $w1 = 0;
  $w0 = 0;
  return $w1, $w0;
end

segment code;
register $u1, $u0, $v1, $v0;
register $w1, $w0;

# test: -1024.0 * 1025.0
$u1 = #C00A800000000000;
$u0 = 0;
```

```
$v1 = #400A802000000000;
$v0 = 0;
$w1, $w0 = call DFmul($u1, $u0, $v1, $v0);

trap 0, HALT, 0;
```

TAOCP 4.3.1 – Classical Multiple-Precision Algorithms

proga: Program A – Multiple-precision addition

proga.mms

```
<<<---
% Author M.Ruckert
 LOC Data_Segment
 GREG @
U OCTA -1,-1,3
V OCTA 1,-1,3
W OCTA 0,0,0,0
n IS 3


 LOC #100
Main LDA $1,U
 LDA $2,V
 LDA $3,W
 SET $4,n
 PUSHJ $0,:Classic:Add
 TRAP 0,Halt,0

 PREFIX :Classic:

U IS $0 Parameter.
V IS $1
W IS $2
n IS $3

j IS $4 Local variables.
k IS $5
wj IS $6
tmp IS $7
--->>>
Add 8ADDU U,n,U; 8ADDU V,n,V; 8ADDU W,n,W $X\is X+8n$
 SL j,n,3; NEG j,j    $j \is -8n$
 SET k,0          $k \is 0$
 JMP 2F
1H LDOU tmp,U,j; ADDU wj,k,tmp  $w_j is u_j+k$
 ZSZ k,wj,k     carry?
 LDOU tmp,V,j; ADDU wj,wj,tmp  $w_j\is w_j+v_j$
 CMPU tmp,wj,tmp; CSN k,tmp,1  carry?
 STOU wj,W,j
 ADD j,j,8
2H PBN j,1B probably $j<0$
 STOU k,W,j    $w_n\is k$
<<<---
 POP 0,0
```

proga.plm

```
# Multi-precision addition left-to-right (TAOCP 4.3.1, Algorithm ...
# W = U + V (n-word arrays).  Returns carry in W[n].
# Parameters: U=$0, V=$1, W=$2, n=$3

# Based on an .mms file by M.Ruckert
def Add($U, $V, $W, $n):
  register $j, $k, $wj, $tmp;

  $U = $U 8addu $n;   # advance to U[n]
  $V = $V 8addu $n;
  $W = $W 8addu $n;
  $j = $n << 3;
  $j = - $j;   # j = -8*n
  $k = 0;

  while likely $j is neg:
    $tmp = $U[$j].o;
    $wj = $k +u $tmp;
    $k ?= $k if $wj is zero;   # carry if wj == 0 (overflow on 0+carry)
    $tmp = $V[$j].o;
    $wj = $wj +u $tmp;
    $tmp = $wj cmpu $tmp;
    $k = 1 if $tmp is neg;   # carry if wj < v_j (overflow on add)
    $W[$j].o = $wj;
    $j = $j + 8;
  end
  $W[$j].o = $k;   # w_n = carry
  return;
end

segment data;
greg $UBase;
octa U -1, -1, 3;
greg $VBase;
octa V 1, -1, 3;
greg $WBase;
octa Wbuf 0, 0, 0, 0;
segment code;
call Add($UBase, $VBase, $WBase, 3);
trap 0, HALT, 0;
```

progac: Program A – Multiple-precision addition with carry

progac.mms

```
<<<---
% Author M.Ruckert
 LOC Data_Segment
 GREG @
U OCTA -1,-1,3
 OCTA 1,-1,3
 OCTA 1,-1,3
 OCTA 1,-1,3
 OCTA 1,-1,3
 OCTA 1,-1,3
m IS 2
W OCTA 0,0,0,0
n IS 3


 LOC #100
Main LDA $1,U
 SET $2,m
 LDA $3,W
 SET $4,n
 PUSHJ $0,:Classic:AddC
 TRAP 0,Halt,0

 PREFIX :Classic:
--->>>
U IS $0 Parameter.
m IS $1
W IS $2
n IS $3

i IS $4 Local variables.
U0 IS $5
j IS $6
k IS $7
wj IS $8
tmp IS $9

AddC SET U0,U; 8ADDU W,n,W   $2$ $X is X+8n$
 SL j,n,3; NEG j,j      $2$ $j is -8n$
 SET k,0            $1$ $k is 0$
 JMP 4F    $1$

1H 8ADDU U,n,U0    $n$
 LDOU tmp,U,j; ADDU wj,k,tmp  $2n$ $w_j is u_{0j}+k$
 ZSZ  k,wj,k    $n$ carry?
 SET i,m    $n$
 JMP 3F    $n$
2H LDOU tmp,U,j; ADDU wj,wj,tmp  $2n(m-1)$ $w_j is w_j+u_{ij}$
 CMPU tmp,wj,tmp; ZSN tmp,tmp,1 $2n(m-1)$ carry?
 ADD k,k,tmp    $n(m-1)$
3H 8ADDU U,n,U    $nm$ Advance $i$.
 SUB i,i,1    $nm$
 PBP i,2B    $n(m+2)$
 STOU wj,W,j    $n$
 ADD j,j,8    $n$
4H PBN j,1B    $n+3$ Probably $j<0$.
 STOU k,W,j    $1$ $w_n is k$
<<<---
 POP 0,0
```

progac.plm

```
# Multi-precision addition with carry array (TAOCP 4.3.1, Algorit...
# W = U[0] + U[1] + ... + U[m-1], each n words.  Parameters: U=$0...

# Based on an .mms file by M.Ruckert
def AddC($U, $m, $W, $n):
  register $i;
  register $U0;
  register $j;
  register $k;
  register $wj;
  register $tmp;

  $U0 = $U;   # save base pointer
  $W = $W 8addu $n;   # W = past end
  $j = $n << 3;
  $j = - $j;
  $k = 0;

  while likely $j is neg:
    $U = $U0 8addu $n;   # U = U0 + n (= &U0[n] = past end of first row)
    $tmp = $U[$j].o;
    $wj = $k +u $tmp;
    $k ?= $k if $wj is zero;   # carry from adding k to U[0][j]
    $i = $m;
    goto InnerCheck;

    InnerBody:
    $U = $U 8addu $n;   # advance to next row
    $tmp = $U[$j].o;
    $wj = $wj +u $tmp;
    $tmp = $wj cmpu $tmp;
    $tmp ?= 1 if $tmp is neg;   # carry from this add
    $k = $k +u $tmp;

    InnerCheck:
    $U = $U 8addu $n;
    $i = $i - 1;
    goto InnerBody if $i is pos;

    $W[$j].o = $wj;
    $j = $j + 8;
  end
  $W[$j].o = $k;
  return;
end

segment data $UBase;
octa U -1, -1, 3,
         1, -1, 3,
         1, -1, 3;
greg $WBase;
octa Wbuf 0, 0, 0, 0;
segment code;
call AddC($UBase, 3, $WBase, 3);
trap 0, HALT, 0;
```

progb: Program B – Multiple-precision subtraction

progb.mms

```
<<<---
% Author M.Ruckert
 LOC Data_Segment
 GREG @
U OCTA -1,-1,3
V OCTA 1,-1,3
W OCTA 0,0,0,0
n IS 3


 LOC #100
Main LDA $1,U
 LDA $2,V
 LDA $3,W
 SET $4,n
 PUSHJ $0,:AddRL:AddRL
 TRAP 0,Halt,0

 PREFIX :AddRL:
U IS $0 Parameter.
V IS $1
W IS $2
n IS $3

j IS $4 Local variables.
i IS $5
w IS $6
tmp IS $7
--->>>
AddRL SL j,n,3                  $1$ B1. $j is n-1$
 STCO 0,W,j   $1$ $w_n is 0$
 SUB j,j,8   $1$ $j is n-1$
2H LDOU w,U,j   $n$ B2.
 LDOU tmp,V,j; ADDU w,w,tmp $n$ $w_j is u_j+v_j mod b$
 STOU w,W,j   $n$
 CMPU tmp,w,tmp  $n$ B3.
 PBNN tmp,4F   $n+2L$
 SET i,j   $L$ $i is j$
0H ADD i,i,8   $K$ $i is i+1$
 LDOU w,W,i   $K$ $w_i is w_i+1 mod b$
 ADDU w,w,1   $K$
 STOU w,W,i   $K$
 BZ w,0B   $3(K-L)-2$ Repeat until $w_i+1<b$.
4H SUB j,j,8   $n$ B4. $j is j-1$
 PBNN j,2B   $n+2$ If $j\ge 0$ go back to B2.
<<<---
 POP 0,0
```

progb.plm

```
# Multi-precision addition right-to-left with carry propagation (...
# W = U + V, propagating carry left.  Parameters: U=$0, V=$1, W=$...

# Based on an .mms file by M.Ruckert
def AddRL($U, $V, $W, $n):
  register $j;
  register $i;
  register $w;
  register $tmp;

  $j = $n << 3;   # j = 8*n - byte offset of W[n-1] from W[0] before adjust
  # B1: W[n] = 0 (zero the carry word)
  $W[$j].o = 0;
  $j = $j - 8;

  # B2: Add words right-to-left (from W[n-1] down to W[0])
  B2:
  $w = $U[$j].o;
  $tmp = $V[$j].o;
  $w = $w +u $tmp;
  $W[$j].o = $w;
  # B3: Check for carry
  $tmp = $w cmpu $tmp;   # if w < v_j (unsigned), overflow happened
  goto B4 if $tmp is nonneg;
  # Propagate carry leftward
  $i = $j;
  CarryLoop:
  $i = $i + 8;
  $w = $W[$i].o;
  $w = $w +u 1;
  $W[$i].o = $w;
  goto CarryLoop if $w is zero;
  # B4: Decrement j
  B4:
  $j = $j - 8;
  goto B2 if $j is nonneg;
  return;
end

segment data $UBase;
octa U -1, -1, 3;
greg $VBase;
octa V 1, -1, 3;
greg $WBase;
octa Wbuf 0, 0, 0, 0;
segment code;
call AddRL($UBase, $VBase, $WBase, 3);
trap 0, HALT, 0;
```

progd: Program D – Multiple-precision division

progd.mms

```
<<<---
% Author M.Ruckert
 LOC Data_Segment
m IS 3
n IS 2
 GREG @
U OCTA 0,0,0,0,1,#FF00FF m + n
V OCTA 12,0,#EE00EE  n
Q OCTA 7,7,7,7,#DD00DD  m+1

 LOC #100
Main LDA $1,U
 LDA $2,V
 LDA $3,Q
 SET $4,m+n
 SET $5,n
 PUSHJ $0,:Classic:Div
 TRAP 0,Halt,0


 PREFIX :LeadingZeros:
 % Count leading zeros see execise 43. in Section $1.3.1^\prime$
x IS     $0 Parameter
magic IS     $1 Local Variables
xm1 IS     $2

:LeadingZeros SETL    magic,#4080
 INCML magic,#1020
 INCMH magic,#0408
 INCH magic,#0102
 MOR x,x,magic; MOR x,magic,x Reverse $x$.
 SUBU xm1,x,1; SADD $0,xm1,x   Count trailing zeros.
 POP 1,0

 PREFIX :ShiftLeft:
 % Shift left array \.X of \.n octabyte by \.d bit. Return last \.{carry}.
X IS $0 Parameter
n IS $1
d IS $2

i IS $1 shares register with \.n
dm IS $3 $64-d$
carry IS $4 Return value
xi IS $5
tmp IS $6

:ShiftLeft NEG dm,64,d         Shift $x$ left $d$ bits.
 SET carry,0
 8ADDU X,n,X; SLU i,n,3; NEG i,i
0H LDOU  xi,X,i                Load $x_i$.
 SLU tmp,xi,d; OR tmp,tmp,carry   Shift and add carry.
 STOU tmp,X,i           Store $x_i$
 SRU carry,xi,dm      New carry.
 ADD i,i,8
 PBN i,0B
 SET $0,carry
 POP 1,0

 PREFIX :ShiftRight:
 % Shift left array \.X of \.n octabyte by \.d bit. Return last \.{carry}.
X IS $0 Parameter
n IS $1
d IS $2

i IS $1 shares register with \.n
dm IS $3 $64-d$
carry IS $4 Return value
xi IS $5
tmp IS $6

:ShiftRight NEG dm,64,d         Shift $x$ right $d$ bits.
 SET carry,0
 8ADDU X,n,X; SLU i,n,3; NEG i,i $i\is 0$
0H LDOU  xi,X,i                 Load $x_i$.
 SRU tmp,xi,d; OR tmp,tmp,carry    Shift and add carry.
 STOU tmp,X,i           Store $x_i$
 SLU carry,xi,dm        New carry.
 ADD i,i,8
 PBN i,0B
 SET $0,carry
 POP 1,0

 PREFIX :Classic:

U IS $0 Parameters.
V IS $1
Q IS $2
nu IS $3
m IS $3 Alias for nu
n IS $4

return IS $5
vn1 IS $6 Local variables.
vn2 IS $7
d IS $8
c IS $9
u IS $10
j IS $11
Uj IS $12
qh IS $13
rh IS $14
p0 IS $15
p1 IS $16
i IS $17
ujn IS $18
t IS $29

rH IS :rH  Import
rD IS :rD
rR IS :rR
```

progd.plm

```
# Multi-precision division: Q = U / V, remainder left in U (TAOCP...
# U is nu words, V is n words, Q is m+1 words where m = nu - n.
# Embedded subroutines: LeadingZeros, ShiftLeft, ShiftRight.

# Count leading zeros in a 64-bit word (Exercise 1.3.1'-43)
# Based on an .mms file by M.Ruckert
def LeadingZeros($x):
  register $magic;
  register $xm1;

  $magic = 0;
  $magic orl=  #4080;
  $magic orml= #1020;
  $magic ormh= #0408;
  $magic orh=  #0102;
  $x = $x mor $magic;
  $x = $magic mor $x;   # reverse x via MOR trick
  $xm1 = $x -u 1;
  $0 = $xm1 sadd $x;   # count trailing zeros of reversed = leading zeros of original
  return $0;
end

# Shift left array X of n octabytes by d bits; return carry
def ShiftLeft($X, $n, $d):
  register $i = $1;# reuses n's register
  register $dm;# 64 - d
  register $carry;
  register $xi;
  register $tmp;

  $dm = - $d;
  $dm = $dm + 64;
  $carry = 0;
  $X = $X 8addu $n;
  $i = $n << 3;
  $i = - $i;
  Loop:
  $xi = $X[$i].o;
  $tmp = $xi <<u $d;
  $tmp = $tmp | $carry;
  $X[$i].o = $tmp;
  $carry = $xi >>u $dm;
  $i = $i + 8;
  goto Loop if $i is neg;
  return $carry;
end

# Shift right array X of n octabytes by d bits; return carry
def ShiftRight($X, $n, $d):
  register $i = $1;# reuses n's register
  register $dm;# 64 - d
  register $carry;
  register $xi;
  register $tmp;

  $dm = - $d;
  $dm = $dm + 64;
  $carry = 0;
  $X = $X 8addu $n;
  $i = $n << 3;
  $i = - $i;
  Loop:
  $xi = $X[$i].o;
  $tmp = $xi >>u $d;
  $tmp = $tmp | $carry;
  $X[$i].o = $tmp;
  $carry = $xi <<u $dm;
  $i = $i + 8;
  goto Loop if $i is neg;
  return $carry;
end

# Main division procedure
# Parameters: U=$0 (nu words), V=$1 (n words), Q=$2 (quotient), n...
def Div($U, $V, $Q, $nu, $n):
  register $ret;
  register $vn1;
  register $vn2;
  register $d;
  register $c;
  register $u;
  register $j;
  register $Uj;
  register $qh;
  register $rem;
  register $p0;
  register $p1;
  register $i;
  register $ujn;
  register $tmp2;
  register $t = $29;
  register $m = $3;# alias nu - overwritten to m = nu - n

  $ret = $rJ;
  $ujn = 0;

  # D1: Normalize V (find leading zeros of V[n-1], shift V and U by...
  $t = $n - 1;
  $t = $t << 3;
  $vn1 = $V[$t].o;   # load V[n-1]
  $0 = call LeadingZeros($vn1);
  $d = $0;
  goto D2 if $d is zero;
  $1 = $V;
  $2 = $n;
  $3 = $d;
  $t = call ShiftLeft($1, $2, $3);
  $1 = $U;
  $2 = $nu;
  $3 = $d;
```

```
rJ IS :rJ
LeadingZeros IS :LeadingZeros
ShiftLeft IS :ShiftLeft
ShiftRight IS :ShiftRight
--->>>
Div GET return,rJ   1
 SET ujn,0    1 D1. Normalize.
 SUB t,n,1; SL t,t,3   2
 LDOU d+1,V,t    1
 PUSHJ d,LeadingZeros   8 See exercise $1.3.1^\prime$-43.
 BZ      d,2F    1
 SET t+1,V; SET t+2,n  2
 SET t+3,d    1
 PUSHJ t,ShiftLeft            See exercise 25.
 SET t+1,U; SET t+2,nu  2
 SET t+3,d    1
 PUSHJ t,ShiftLeft    See exercise 25.
 SET ujn,t    1 Keep carry.

2H 8ADDU V,n,V    1 D2. Initialize $j$.
 NEG  t,8; LDOU vn1,V,t  2
 NEG  t,16; LDOU vn2,V,t  2
 SUB m,nu,n    1 Get $m$ from the size of $u$ and $v$
 SET j,m    1
 8ADDU Uj,j,U; 8ADDU Uj,n,Uj  2 $U_j is U+8(j+n)$
 JMP 0F    1 Skip loading $u_{j+n}$.

3H LDOU ujn,Uj,0   $m$ D3. Calculate $\hat{q$.}
0H NEG t,8; LDOU u,Uj,t  $2(m+1)$ Get $u_{j+n-1}$
 CMPU t,ujn,vn1   $m+1$ If $u_{j+n}\ge v_{n-1}$ then $\hat{q}\ge b$
 BN t,Small    $m+1$
Large SUB ujn,ujn,vn1   $m+1$ Case $u_{j+n}\ge v_{n-1}$
 PUT rD,ujn    $m+1$
 DIVU qh,u,vn1; SET qh,0  $61(m+1)$ $\hat{q} is b$
 GET rh,rR    $m+1$
 JMP Adjust    $m+1$

Small PUT rD,ujn    $m+1$ Case $u_{j+n} < v_{n-1}$
 DIVU qh,u,vn1   $60(m+1)$ $\hat{q} is \lfloor (u_{j+n}b+u_{j+n-1})/v_{n-1}\rfloor$
 GET rh,rR    $m+1$ $\hat{r} is \cdots mod v_{n-1}$

Test MULU p0,qh,vn2   $10(m+1)$ $p_1b+p_0 is \hat{q}v_{n-2}$
 GET p1,rH    $m+1$
 CMPU t,p1,rh    $m+1$ Compare high 64 bits.
 BN t,4F; BP t,Adjust   $m+1$
 NEG t,16; LDOU u,Uj,t   $2(m+1)$ Load $u_{j+n-2}$.
 CMPU t,p0,u; BN t,4F   $2(m+1)$ Compare low 64 bits.
Adjust SUBU qh,qh,1    $m+1$ Decrease $\hat{q}$.
 ADDU rh,rh,vn1   $m+1$ Tentatively add $\hat{r}+ v_{n-1}$
 CMPU t,rh,vn1   $m+1$ Check if overflow
 BNN t,Test    $m+1$ If not, repeat the test.
     %   & D4. Multiply and subtract.
4H SL i,n,3; NEG i,i      $2(m+1)$ $i is -8n$
 SET c,0          $m+1$ $carry is 0$
0H LDOU t,Uj,i    $n(m+1)$ Load $u_{j+i}$.
       SUBU  u,t,c    $n(m+1)$ $u_{j+i} is u_{j+i}-carry$
 CMPU t,t,c; ZSN c,t,1  $2n(m+1)$ Carry?
 LDOU t,V,i    $n(m+1)$
 MULU p0,t,qh    $10n(m+1) $
 CMPU t,u,p0; CSN c,t,1  $n(m+1)$ Carry?
 SUBU  u,u,p0    $n(m+1)$ $u_{j+i} is u_{j+i}-\hat{q}v_i$
 GET p1,rH; ADDU c,c,p1       $2n(m+1)$ Carry from the multiplication
 STOU u,Uj,i    $n(m+1)$
 ADD i,i,8    $n(m+1)$
 PBN i,0B    $n(m+1)$ Repeat for $0\le i < n$

 SUBU  u,ujn,c    $m+1$ Complete D4 for $i=n$.
 CMPU t,ujn,c; ZSN c,t,1  $2(m+1)$
 CMP t,j,m; BNN t,0F   $1(m+1)$ Store unless $j=m$
 STOU u,Uj,i    $m+1$ $u{j+n} is u_{j+i}+0+carry$

0H PBZ c,5F    $m+1$ D5. Test remainder.

 SUB qh,qh,1     D6. Add back.
 SL t,j,3
 STOU qh,Q,j
 SL i,n,3; NEG i,i       $i is -8n$
 SET c,0          $carry is 0$
0H LDOU u,Uj,i; ADDU u,u,c   $u_{j+i} is u_{j+i}+carry$
 ZSZ  c,u,c    Carry?
 LDOU t,V,i; ADDU u,u,t   $u_{j+i} is u_{j+i}+v_i$
 CMPU t,u,t; CSN c,t,1   Carry?
 STOU u,Uj,i
 ADD i,i,8
 PBN i,0B     Probably $j<0$.
 LDOU u,Uj,i; ADDU u,u,c
 STOU u,Uj,i     $u{j+n} is u_{j+i}+0+carry$
 JMP 7F

5H SL t,j,3    $m+1$ D5. Test remainder. (continued)
 STOU qh,Q,t    $m+1$ $q_j is \hat{q}$

7H SUB j,j,1    $m+1$ D7. Loop on $j$.
 SUB Uj,Uj,8    $m+1$
 PBNN j,3B    $m+1$
 BZ d,8F    1
 SET t+1,U    1 D8. Unnormalize.
 SET t+2,n    1
 SET t+3,d    1
 PUSHJ t,ShiftRight   \ See exercise 26.
8H IS @
<<<---
 PUT rJ,return
 POP 0,0
```

```
$t = call ShiftLeft($1, $2, $3);
$ujn = $t;

D2:
$V = $V 8addu $n;
$t = -8;
$vn1 = $V[$t].o;   # V[n-1] after normalization
$t = -16;
$vn2 = $V[$t].o;   # V[n-2]
$m = $nu -u $n;   # nu becomes m
$j = $m;
$Uj = $Uj 8addu $j;
$Uj = $Uj 8addu $n;   # Uj = &U[j+n]
goto D2loop;

D3:
$ujn = [$Uj].o;   # load u_{j+n}
D2loop:
$t = -8;
$u = $Uj[$t].o;   # u_{j+n-1}
$t = $ujn cmpu $vn1;
goto Small if $t is neg;
# Large: u_{j+n} >= v_{n-1}
Large:
$ujn = $ujn -u $vn1;
$rD = $ujn;
$qh = $u /$u $vn1;
$qh = 0;   # quotient is b = 2^64 on overflow; set max
$rem = $rR;
goto Adjust;

Small:
$rD = $ujn;
$qh = $u /$u $vn1;   # qh = floor((u_{j+n}*b + u_{j+n-1}) / v_{n-1})
$rem = $rR;

Test:
$p0 = $qh *$u $vn2;   # (p1,p0) = qh * v_{n-2}
$p1 = $rH;
$t = $p1 cmpu $rem;
goto D4 if $t is neg;
goto Adjust if $t is pos;
$t = -16;
$u = $Uj[$t].o;   # u_{j+n-2}
$t = $p0 cmpu $u;
goto D4 if $t is neg;
Adjust:
$qh = $qh -u 1;
$rem = $rem +u $vn1;
$t = $rem cmpu $vn1;
goto Test if $t is nonneg;

# D4: Multiply and subtract
D4:
$i = $n << 3;
$i = - $i;
$c = 0;
InnerLoop:
$t = $Uj[$i].o;
$u = $t -u $c;
$tmp2 = $t cmpu $c;
$c = 1 if $tmp2 is neg;   # borrow from subtracting the incoming carry
$t = $V[$i].o;
$p0 = $t *$u $qh;
$tmp2 = $u cmpu $p0;
$c = 1 if $tmp2 is neg;
$u = $u -u $p0;
$tmp2 = $rH;
$c = $c +u $tmp2;
$Uj[$i].o = $u;
$i = $i + 8;
goto InnerLoop if $i is neg;
$u = $ujn -u $c;
$t = $ujn cmpu $c;
$c ?= 1 if $t is neg;
$t = $j cmpu $m;
goto SkipStore if $t is nonneg;
$Uj[$i].o = $u;

SkipStore:
goto D5 if $c is zero;
# D6: Add back
$qh = $qh -u 1;
$t = $j << 3;
$Q[$t].o = $qh;
$i = $n << 3;
$i = - $i;
$c = 0;
AddLoop:
$u = $Uj[$i].o;
$u = $u +u $c;
$c ?= $c if $u is zero;
$t = $V[$i].o;
$u = $u +u $t;
$t = $u cmpu $t;
$c = 1 if $t is neg;
$Uj[$i].o = $u;
$i = $i + 8;
goto AddLoop if $i is neg;
$u = $Uj[$i].o;
$u = $u +u $c;
$Uj[$i].o = $u;
goto D7;

D5:
$t = $j << 3;
$Q[$t].o = $qh;

D7:
$j = $j - 1;
$Uj = $Uj - 8;
goto D3 if $j is nonneg;
goto D8 if $d is zero;
# D8: Unnormalize
$1 = $U;
$2 = $n;
```

```
  $3 = $d;
  $t = call ShiftRight($1, $2, $3);

  D8:
  $rJ = $ret;
  return;
end

segment data $UBase;
octa U 0, 0, 0, 0, 1, #FF00FF;
greg $VBase;
octa V 12, 0, #EE00EE;
greg $QBase;
octa Qbuf 7, 7, 7, 7;
segment code;
call Div($UBase, $VBase, $QBase, 6, 3);
trap 0, HALT, 0;
```

progm: Program M – Multiple-precision multiplication

progm.mms

```
<<<---
% Author M.Ruckert
 LOC Data_Segment
 GREG @
U OCTA 2,0,1,#FF00FF

W OCTA 7,7,7,7,7,7,#DD00DD
m IS 3
n IS 2


 LOC #100
Main LDA $1,U
 SET $2,3
 LDA $3,W
 SET $4,m
 SET $5,n
 PUSHJ $0,:Classic:Mul
 TRAP 0,Halt,0

 PREFIX :Classic:

U IS $0 Parameter.
V IS $1
W IS $2
Wj IS $2
m IS $3
n IS $4

i IS $5 Local variables.
j IS $6
k IS $7
wj IS $8
vj IS $9
t IS $10
tmp IS $11
rH IS :rH
--->>>
Mul 8ADDU U,m,U; 8ADDU V,n,V
 8ADDU Wj,m,W        Address of $w_{j+m}$
 SL i,m,3; NEG i,i  M1. Initialize.
0H STCO 0,Wj,i       $(w_{m-1}... w_0) is (0...0)$
 ADD i,i,8
 PBN i,0B

 SL j,n,3; NEG j,j  $j \is -8n$
2H LDOU vj,V,j      M2. Zero multiplier?
 PBNZ vj,3F
 STCO 0,Wj,0   $w_{j+m}\is 0$
 JMP 6F

3H SL i,m,3; NEG i,i  M3. Initialize i.
 SET k,0
     %M4. Multiply and add.
4H LDOU t,U,i   $t is u_i$
 MULU t,t,vj   $t\is u_i v_i$
 ADDU  t,t,k   $t\is u_i v_i + k$
 CMPU tmp,t,k; ZSN    k,tmp,1 Carry?
 LDOU tmp,Wj,i; ADDU t,t,tmp $t is u_i v_i + k+ w_{j+i}$
 CMPU tmp,t,tmp; CSN  k,tmp,1 Carry?
 STOU t,Wj,i      $w_{i+j}\is t \bmod b$
 GET tmp,rH; ADDU k,k,tmp $k is \lfloor t/b \rfloor$
 ADD i,i,8   M5. Loop on $i$.
 PBN i,4B

 STOU k,Wj,0   $w_{j+m}\is k$
6H ADD j,j,8   M5. Loop on $j$.
 ADD Wj,Wj,8
 PBN j,2B
 STOU k,Wj,0   $w_{m+j}\is k$
<<<---
 POP 0,0
```

progm.plm

```
# Multi-precision multiplication: W = U * V (TAOCP 4.3.1, Algorit...
# U is m words, V is n words, W is m+n words.
# Parameters: U=$0, V=$1, W=$2, Wj=$2 (alias), m=$3, n=$4

# Based on an .mms file by M.Ruckert
def Mul($U, $V, $Wj, $m, $n):
  register $i;
  register $j;
  register $k;
  register $wj;
  register $vj;
  register $t;
  register $tmp;

  $U = $U 8addu $m;
  $V = $V 8addu $n;
  $Wj = $Wj 8addu $m;   # Wj = address of W[m]

  # M1: Initialize W[0..m-1] = 0
  $i = $m << 3;
  $i = - $i;
  while likely $i is neg:
    $Wj[$i].o = 0;
    $i = $i + 8;
  end

  $j = $n << 3;
  $j = - $j;

  OuterLoop:
  $vj = $V[$j].o;   # load v_j
  goto M3 if $vj is nonzero;
  [$Wj].o = 0;   # w_{j+m} = 0
  goto M6;

  M3:
  $i = $m << 3;
  $i = - $i;
  $k = 0;

  while likely $i is neg:
    $t = $U[$i].o;
    $t = $t *u $vj;   # t = u_i * v_j; high bits in $rH
    $t = $t +u $k;
    $tmp = $t cmpu $k;
    $k ?= 1 if $tmp is neg;   # carry from add
    $tmp = $Wj[$i].o;
    $t = $t +u $tmp;
    $tmp = $t cmpu $tmp;
    $k = 1 if $tmp is neg;   # carry from second add
    $Wj[$i].o = $t;
    $tmp = $rH;
    $k = $k +u $tmp;   # k += high 64 bits of multiply
    $i = $i + 8;
  end
  [$Wj].o = $k;

  M6:
  $j = $j + 8;
  $Wj = $Wj + 8;
  goto OuterLoop if $j is neg;
  [$Wj].o = $k;
  return;
end

segment data $UBase;
octa U 2, 0, 1, #FF00FF;
greg $VBase;
octa V 0, 0, 3;
greg $WBase;
octa Wbuf 7, 7, 7, 7, 7, 7, 7;
segment code;
call Mul($UBase, $VBase, $WBase, 4, 3);
trap 0, HALT, 0;
```

progms: Single-digit multiple-precision multiplication

progms.mms

```
<<<---
% Author M.Ruckert
 LOC Data_Segment
 GREG @
U OCTA 2,0,1,#FF00FF
v IS 3
W OCTA 7,7,7,7,7,7,#DD00DD
m IS 3


 LOC #100
Main LDA $1,U
 SET $2,v
 LDA $3,W
 SET $4,m
 PUSHJ $0,:Classic:MulS
 TRAP 0,Halt,0

 PREFIX :Classic:

U IS $0 Parameter.
v IS $1
W IS $2
m IS $3


i IS $4 Local variables.
k IS $5
t IS $6
tmp IS $7
rH IS :rH
--->>>
MulS 8ADDU U,m,U   $1$
 8ADDU W,m,W        $1$ Address of $w_{m}$
 SL i,m,3; NEG i,i  $2$ $i is m$
 SET k,0   $1$
4H LDOU t,U,i   $m$ $t is u_i$
 MULU t,t,v   $10m$ $t is u_i v$
 ADDU  t,t,k   $m$ $t is u_i v + k$
 CMPU tmp,t,k; ZSN k,tmp,1 $2m$ $k is carry$
 STOU t,W,i      $m$ $w_i is t mod b$
 GET tmp,rH; ADDU k,k,tmp $2m$ $k is \lfloor t/b \rfloor + k$
 ADD i,i,8   $m$ $i is i-1$
 PBN i,4B   $m+2$
 STOU k,W,0   $1$ $w_m is k$
<<<---
 POP 0,0
```

progms.plm

```
# Multi-precision multiply by single word: W = U * v (TAOCP 4.3.1...
# Parameters: U=$0 (m-word array), v=$1 (scalar), W=$2 (result, m...

# Based on an .mms file by M.Ruckert
def MulS($U, $v, $W, $m):
  register $i;
  register $k;
  register $t;
  register $tmp;

  $U = $U 8addu $m;   # U = U + 8*m = address of U[m]
  $W = $W 8addu $m;   # W = W + 8*m = address of W[m]
  $i = $m << 3;
  $i = - $i;   # i = -8*m
  $k = 0;

  while likely $i is neg:
    $t = $U[$i].o;
    $t = $t *u $v;   # t = u_i * v; high bits in $rH
    $t = $t +u $k;   # t += carry k
    $tmp = $t cmpu $k;
    $k ?= 1 if $tmp is neg;   # carry from addition
    $W[$i].o = $t;
    $tmp = $rH;
    $k = $k +u $tmp;   # k += high bits of multiply
    $i = $i + 8;
  end
  [$W].o = $k;   # w_m = final carry
  return;
end

segment data $UBase;
octa U 2, 0, 1, #FF00FF;
greg $WBase;
octa W 7, 7, 7, 7, 7, 7;
segment code;
call MulS($UBase, 3, $WBase, 4);
trap 0, HALT, 0;
```

progs: Program S – Multiple-precision subtraction (variant)

progs.mms

```
<<<---
% Author M.Ruckert
 LOC Data_Segment
 GREG @
U OCTA 0,0,1
V OCTA 1,0,0
W OCTA 0,0,0,0
n IS 3


 LOC #100
Main LDA $1,U
 LDA $2,V
 LDA $3,W
 SET $4,n
 PUSHJ $0,:Classic:Sub
 TRAP 0,Halt,0

 PREFIX :Classic:

U IS $0 Parameter.
V IS $1
W IS $2
n IS $3

j IS $4 Local variables.
k IS $5
wj IS $6
vj IS $7
uj IS $8
tmp IS $9
--->>>
Sub 8ADDU U,n,U; 8ADDU V,n,V; 8ADDU W,n,W $X\is X+8n$
 SL j,n,3; NEG j,j    $j \is -8n$
 SET k,0          $k \is 0$
 JMP 2F
1H LDOU uj,U,j
 SUBU  wj,uj,k    $w_j\is u_j-k$
 ZSZ  k,uj,k    carry?
 LDOU vj,V,j
 CMPU tmp,wj,vj; CSN k,tmp,1  carry?
 SUBU  wj,wj,vj   $w_j\is w_j-v_j$
 STOU wj,W,j
 ADD j,j,8
2H PBN j,1B    probably $j<0$
 BNZ k,Error    $k\ne 0$ if $u<v$
<<<---
 POP 0,0

Error SET $255,1
 TRAP 0,:Halt,0
--->>>
```

progs.plm

```
# Multi-precision subtraction: W = U - V (TAOCP 4.3.1, Algorithm ...
# Exits with error if U < V.  Parameters: U=$0, V=$1, W=$2, n=$3

# Based on an .mms file by M.Ruckert
def Sub($U, $V, $W, $n):
  register $j, $k, $wj, $vj, $uj, $tmp;

  $U = $U 8addu $n;
  $V = $V 8addu $n;
  $W = $W 8addu $n;
  $j = $n << 3;
  $j = - $j;
  $k = 0;

  while likely $j is neg:
    $uj = $U[$j].o;
    $wj = $uj -u $k;
    $k ?= $k if $uj is zero;   # carry if uj==0 and k==1
    $vj = $V[$j].o;
    $tmp = $wj cmpu $vj;
    $k = 1 if $tmp is neg;   # borrow if wj < vj unsigned
    $wj = $wj -u $vj;
    $W[$j].o = $wj;
    $j = $j + 8;
  end
  return if $k is zero;
  # Error: U < V (underflow)
  $255 = 1;
  trap 0, HALT, 0;
end

segment data $UBase;
octa U 0, 0, 1;
greg $VBase;
octa V 1, 0, 0;
greg $WBase;
octa Wbuf 0, 0, 0, 0;
segment code;
call Sub($UBase, $VBase, $WBase, 3);
trap 0, HALT, 0;
```

shiftleft: Shift multiple-precision number left

shiftleft.mms

```
<<<---
% Author M.Ruckert
 LOC Data_Segment
m IS 3
n IS 2
 GREG @
U OCTA 0,0,0,0,1,#FF00FF m + n
V OCTA 12,0,#EE00EE  n
Q OCTA 7,7,7,7,#DD00DD  m+1

 LOC #100
Main LDA $1,U
 LDA $2,V
 LDA $3,Q
 SET $4,m+n
 SET $5,n
 PUSHJ $0,:ShiftLeft
 TRAP 0,Halt,0


--->>>
 PREFIX :ShiftLeft:
 % Shift left array \.X of \.n octabyte by \.d bit. Return last \.{carry}.
X IS $0 Parameter
n IS $1
d IS $2

i IS $1 shares register with \.n
dm IS $3 $64-d$
carry IS $4 Return value
xi IS $5
tmp IS $6

:ShiftLeft NEG dm,64,d         Shift $x$ left $d$ bits.
 SET carry,0
 8ADDU X,n,X; SLU i,n,3; NEG i,i
0H LDOU  xi,X,i                Load $x_i$.
 SLU tmp,xi,d; OR tmp,tmp,carry   Shift and add carry.
 STOU tmp,X,i           Store $x_i$
 SRU carry,xi,dm      New carry.
 ADD i,i,8
 PBN i,0B
 SET $0,carry
 POP 1,0
```

shiftleft.plm

```
# Shift left array X of n octabytes by d bits; return last carry ...
# Parameters: X=$0 (array address), n=$1 (number of octabytes), d...

# Based on an .mms file by M.Ruckert
def ShiftLeft($X, $n, $d):
  register $i = $1;# reuses n's register after computing start address
  register $dm;# 64 - d
  register $carry;
  register $xi;
  register $tmp;

  $dm = - $d;
  $dm = $dm + 64;
  $carry = 0;
  $X = $X 8addu $n;   # X = address of X[n] = one past end
  $i = $n << 3;
  $i = - $i;   # i = -8*n - negative byte offset from X[n] to X[0]

  while likely $i is neg:
    $xi = $X[$i].o;
    $tmp = $xi <<u $d;
    $tmp = $tmp | $carry;   # shift xi left, OR in previous carry
    $X[$i].o = $tmp;
    $carry = $xi >>u $dm;   # new carry = bits shifted out
    $i = $i + 8;
  end
  return $carry;
end

segment data $XBase;
octa U 0, 0, 0, 0, 1, #FF00FF;
segment code;
$0 = call ShiftLeft($XBase, 6, 3);
trap 0, HALT, 0;
```

shiftright: Shift multiple-precision number right

shiftright.mms

```
<<<---
% Author M.Ruckert
 LOC Data_Segment
m IS 3
n IS 2
 GREG @
U OCTA 0,0,0,0,1,#FF00FF m + n
V OCTA 12,0,#EE00EE  n
Q OCTA 7,7,7,7,#DD00DD  m+1

 LOC #100
Main LDA $1,U
 LDA $2,V
 LDA $3,Q
 SET $4,m+n
 SET $5,n
 PUSHJ $0,:ShiftRight
 TRAP 0,Halt,0

--->>>
 PREFIX :ShiftRight:
 % Shift left array \.X of \.n octabyte by \.d bit. Return last \.{carry}.
X IS $0 Parameter
n IS $1
d IS $2

i IS $1 shares register with \.n
dm IS $3 $64-d$
carry IS $4 Return value
xi IS $5
tmp IS $6

:ShiftRight NEG dm,64,d         Shift $x$ right $d$ bits.
 SET carry,0
 8ADDU X,n,X; SLU i,n,3; NEG i,i $i\is 0$
0H LDOU  xi,X,i                 Load $x_i$.
 SRU tmp,xi,d; OR tmp,tmp,carry    Shift and add carry.
 STOU tmp,X,i           Store $x_i$
 SLU carry,xi,dm        New carry.
 ADD i,i,8
 PBN i,0B
 SET $0,carry
 POP 1,0
```

shiftright.plm

```
# Shift right array X of n octabytes by d bits; return last carry...
# Parameters: X=$0 (array address), n=$1 (number of octabytes), d...

# Based on an .mms file by M.Ruckert
def ShiftRight($X, $n, $d):
  register $i = $1;# reuses n's register after computing start address
  register $dm;# 64 - d
  register $carry;
  register $xi;
  register $tmp;

  $dm = - $d;
  $dm = $dm + 64;
  $carry = 0;
  $X = $X 8addu $n;   # X = address past end
  $i = $n << 3;
  $i = - $i;   # i = -8*n

  while likely $i is neg:
    $xi = $X[$i].o;
    $tmp = $xi >>u $d;
    $tmp = $tmp | $carry;   # shift xi right, OR in previous carry
    $X[$i].o = $tmp;
    $carry = $xi <<u $dm;   # new carry = bits shifted out
    $i = $i + 8;
  end
  return $carry;
end

segment data $XBase;
octa U 0, 0, 0, 0, 1, #FF00FF;
segment code;
$0 = call ShiftRight($XBase, 6, 3);
trap 0, HALT, 0;
```

TAOCP 4.4 – Radix Conversion (Exercise 13)

ex13: Binary-to-decimal conversion (exercise 13)

ex13.mms

```
<<<---
% Author M.Ruckert
 LOC Data_Segment
 GREG @
In OCTA -1,-1,-1,-1,-1,#3FFFFFFFFFFFFFFF,#8000000000000000,#FF00FF
Out BYTE 0

 LOC #100
Main LDA $1,In
 SET $2,6
 LDA $3,Out
 PUSHJ $0,Print
 LDA $255,Out
 TRAP 0,Fputs,StdOut
 TRAP 0,Halt,0
--->>>
U IS $0 Parameter
m IS $1
V IS $2
i IS $3 Local variables.
block IS $4
line IS $5
k IS $6
u IS $7
tmp IS $8
v GREG 10*10*10*10*10*10*10*10*10

Print 8ADDU U,m,U
 SET line,2
 SET tmp,'.'   Start first line with a decimal point.
1H STBU tmp,V; INCL V,1
 SET block,7
2H SL i,m,3; NEG i,i  $i is m$
 SET k,0
0H LDOU u,U,i   $u is u_i$
 MULU u,u,v   $u\is u_i v$
 ADDU  u,u,k   $u\is u_i v + k$
 STOU u,U,i      $u_i\is u \bmod b$
 CMPU tmp,u,k; ZSN k,tmp,1 $k\is carry$
 GET tmp,rH; ADDU k,k,tmp $k\is \lfloor t/b \rfloor + k$
 ADD i,i,8   $i \is i-1$
 PBN i,0B

 SLU u,k,32; ADD u,u,v; DIVU u,u,v $u\is\lfloor (2^{32}u+10^9)/10^9\rfloor$
 SET i,9
0H 4ADDU u,u,u; SLU u,u,1    $u is 10u$
 SRU tmp,u,32   Next digit is $\lfloor 10u \rfloor$.
 ADD tmp,tmp,'0'   Convert to ASCII code.
 STB tmp,V,0; INCL V,1
 ANDNMH u,#FFFF   $u\is u \bmod 2^{32}$
 SUB i,i,1
 PBP i,0B   Repeat 9 times.

 SUB block,block,1
 SET tmp,' '; STBU tmp,V; INCL V,1
 BP block,2B  Next block.
 SET tmp,10; STBU tmp,V; INCL V,1
 SET tmp,' '   Start next line with a space.
 SUB line,line,1
 BP line,1B   Next line.
 SET  tmp,0; STBU tmp,V Terminate string with a zero byte.
 POP 0,0
```

ex13.plm

```
# Print multi-precision number as decimal (TAOCP 4.4, Exercise 13...
# Each iteration multiplies the multi-word number by 10^9 and ext...
# Parameters: U=$0 (array, m words), m=$1, V=$2 (output buffer)

# Based on an .mms file by M.Ruckert
greg $v9 = 1000000000;   # 10^9 - multiply factor and digit extractor

def Print($U, $m, $V):
  register $i, $block, $line, $k, $u, $tmp;

  $U = $U 8addu $m;   # U = address of U[m] (one past end)
  $line = 2;
  $tmp = 46;   # '.' - ASCII period
  # Start first line with a decimal point
  NextLine:
  [$V].bu = $tmp;
  $V = $V +u 1;
  $block = 7;

  NextBlock:
  $i = $m << 3;
  $i = - $i;   # i = -8*m - start from U[-m] = U[0]
  $k = 0;

  MulLoop:
  $u = $U[$i].o;   # load u_i
  $u = $u *$u $v9;   # u = u_i * 10^9 - high bits in $rH
  $u = $u +u $k;   # u += carry
  $U[$i].o = $u;   # store u_i mod 2^64
  $tmp = $u cmpu $k;
  $k ?= 1 if $tmp is neg;   # carry if u < old carry (overflow)
  $tmp = $rH;
  $k = $k +u $tmp;   # k += high bits
  $i = $i + 8;
  goto MulLoop if $i is neg;

  # Extract 9 decimal digits from carry k using scaled trick
  $k = $k <<u 32;
  $k = $k + $v9;
  $k = $k /$u $v9;   # k = floor((carry * 2^32 + 10^9) / 10^9)
  $i = 9;
  DigitLoop:
  $k = $k 4addu $k;   # k = 5k
  $k = $k <<u 1;   # k = 10k
  $tmp = $k >>u 32;   # digit = high 32 bits
  $tmp = $tmp + 48;   # to ASCII
  [$V].b = $tmp;
  $V = $V +u 1;
  $k andnmh= #FFFF;   # clear high 32 bits of k
  $i = $i - 1;
  goto DigitLoop if $i is pos;

  $block = $block - 1;
  $tmp = 32;   # ' '
  [$V].bu = $tmp;
  $V = $V +u 1;
  goto NextBlock if $block is pos;
  $tmp = 10;   # newline
  [$V].bu = $tmp;
  $V = $V +u 1;
  $tmp = 32;   # space for next line
  $line = $line - 1;
  goto NextLine if $line is pos;
  $tmp = 0;
  [$V].bu = $tmp;   # null-terminate
  return;
end

segment data $DataBase;
octa In -1, -1, #3FFFFFFFFFFFFFFF;
byte Out 0;
segment code;
$1 = $DataBase;
$2 = 3;
$3 = $DataBase;
$3 = $3 + 24;   # Out is after In (3*8=24 bytes)
call Print($1, $2, $3);
trap 0, HALT, 0;
```

example19: Radix conversion – example 19

example19.mms

```
<<<---
% Author M.Ruckert
 LOC Data_Segment
 GREG @
Decimal BYTE "12345678"
 LOC #100
Main IS @
U IS $0
tmp IS $1
--->>>
ascii GREG #0F0F0F0F0F0F0F0F
m1 GREG #0F000F000F000F00
m2 GREG #FFFF0000FFFF0000
m3 GREG #FFFFFFFF00000000
 LDO U,Decimal
 AND U,U,ascii
 AND tmp,U,m1
 SUBU U,U,tmp
 4ADDU tmp,tmp,tmp
 SRU tmp,tmp,8-1
 ADD U,U,tmp

 AND tmp,U,m2
 SUBU U,U,tmp
 4ADDU tmp,tmp,tmp
 4ADDU tmp,tmp,tmp
 SRU tmp,tmp,16-2
 ADD U,U,tmp

 AND tmp,U,m3
 SUBU U,U,tmp
 4ADDU tmp,tmp,tmp
 4ADDU tmp,tmp,tmp
 4ADDU tmp,tmp,tmp
 4ADDU tmp,tmp,tmp
 SRU tmp,tmp,32-4
 ADD U,U,tmp
```

example19.plm

```
# Binary to decimal using scaled-floor multiplication (TAOCP 4.4,...
# $x = ceil(2^64/10).  Extract digits as $x = floor(u/10 * 2^64).

# Based on an .mms file by M.Ruckert
segment data;
greg $x = 1844674407370955162;   # ceil(2^64 / 10) = 1 + floor((2^63)/5)
greg $DataBase;
byte Decimal 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
octa Binary 9999999999999999998;
segment code;

register $j, $U, $u, $ux, $r, $tmp;

$U = $DataBase;
$u = Binary[0].o;
$j = 0;

while likely $u is pos:
  $tmp = $u *u $x;   # ($rH, .) = u * ceil(2^64/10)
  $ux = $rH;   # ux = floor(u/10) approx
  $tmp = $ux 4addu $ux;   # tmp = 5*ux
  $tmp = $tmp <<u 1;   # tmp = 10*ux
  $r = $u -u $tmp;   # r = u - 10*floor(u/10) = u mod 10 (or 1 too large)
  goto Store if $r is nonneg;
  $ux = $ux -u 1;   # adjust estimate down by 1
  $r = $r + 10;   # r was negative signed, add 10 to get true mod
  Store:
  $U[$j].bu = $r;   # U[j] = digit
  $u = $ux;
  $j = $j + 1;
end

trap 0, HALT, 0;
```

example8: Radix conversion – example 8

example8.mms

```
<<<---
% Author M.Ruckert
 LOC Data_Segment
 GREG @
Decimal BYTE 0
 LOC Decimal+32
Binary OCTA 9999999999999999998

 LOC #100
Main IS @
j IS $0
U IS $1
u IS $2
ux IS $3
r IS $4
tmp IS $5
 LDA U,Decimal
 LDO u,Binary
--->>>
x GREG 1+(1<<63)/5 $x is \lceil2^{64}/10\rceil$
 SET j,0   Set $j\is 0$.
Loop MULU tmp,u,x   $(rH,\cdot) is u\lfloor 2^{64}/10\rfloor$
 GET ux,rH   $\lfloor ux\rfloor \is u \lfloor 2^{64}/10\rfloor/2^{64}$
 4ADDU tmp,ux,ux; SLU tmp,tmp,1  $\.{tmp} \is 10\lfloor ux\rfloor$
 SUBU r,u,tmp     $r \is u -10\lfloor ux\rfloor$
 PBNN r,OF
 SUBU ux,ux,1    Can occur only on first iteration.
 ADD r,r,10
OH STBU  r,U,j   $U_j is r = u mod 10$
 SET u,ux
 ADD j,j,1    $j \is j+1$
 PBP u,Loop   Repeat until result is zero
```

example8.plm

```
# Decimal to binary: 8-digit parallel conversion using bit tricks...
# Each byte holds one ASCII decimal digit. Result packed into one...

# Based on an .mms file by M.Ruckert
segment data;
greg $ascii = #0F0F0F0F0F0F0F0F;
greg $m1    = #0F000F000F000F00;
greg $m2    = #FFFF0000FFFF0000;
greg $m3    = #FFFFFFFF00000000;
greg $DataBase;
byte Decimal #31, #32, #33, #34, #35, #36, #37, #38;   # "12345678" in ASCII
segment code;

register $U, $tmp;

$U = Decimal[0].o;   # load 8 ASCII digits packed into U
$U = $U & $ascii;   # mask to low nibble: each byte now 0..9

# Collapse pairs: each pair of bytes -> one 2-digit value in a by...
$tmp = $U & $m1;
$U = $U -u $tmp;
$tmp = $tmp 4addu $tmp;   # 4*tmp
$tmp = $tmp >>u 7;   # >> 7 = /128, extracts top byte of pair times 10
$U = $U +u $tmp;   # U now has 2-digit values in bytes 0,2,4,6

# Collapse pairs of 2-digit values -> 4-digit values
$tmp = $U & $m2;
$U = $U -u $tmp;
$tmp = $tmp 4addu $tmp;
$tmp = $tmp 4addu $tmp;   # 100*tmp
$tmp = $tmp >>u 14;
$U = $U +u $tmp;

# Collapse pairs of 4-digit values -> 8-digit value
$tmp = $U & $m3;
$U = $U -u $tmp;
$tmp = $tmp 4addu $tmp;
$tmp = $tmp 4addu $tmp;
$tmp = $tmp 4addu $tmp;
$tmp = $tmp 4addu $tmp;   # 10000*tmp
$tmp = $tmp >>u 28;
$U = $U +u $tmp;   # U now holds the 8-digit decimal value

trap 0, HALT, 0;
```

method1a: Radix conversion – right-to-left, method 1a

method1a.mms

```
<<<---
% Author M.Ruckert
 LOC Data_Segment
 GREG @
Decimal BYTE 0
 LOC Decimal+32
Binary OCTA 1234567890

 LOC #100
Main IS @
j IS $0
U IS $1
u IS $2

tmp IS $3
 LDA U,Decimal
 LDO u,Binary
--->>>
 SET j,0   Set $j\is 0$.
1H DIV u,u,10  $rR is u mod 10$ and $u is \lfloor u/10\rfloor$
 GET tmp,rR; STBU tmp,U,j $U_j\is u \bmod 10$
 ADD j,j,1    $j \is j+1$
 PBP u,1B   Repeat until result is zero
```

method1a.plm

```
# Binary to decimal conversion by repeated division (TAOCP 4.4, M...
# Output: decimal digits stored least-significant first in U[0], ...

# Based on an .mms file by M.Ruckert
segment data $DataBase;
byte Decimal 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
octa Binary 1234567890;
segment code;

register $j, $U, $u, $tmp;

$U = $DataBase;   # U = address of Decimal buffer
$u = Binary[0].o;   # u = Binary value
$j = 0;

while likely $u is pos:
  $u = $u / 10;   # quotient in u, remainder in $rR
  $tmp = $rR;
  $U[$j].bu = $tmp;   # U[j] = u mod 10
  $j = $j + 1;
end

trap 0, HALT, 0;
```

method1b: Radix conversion – right-to-left, method 1b

method1b.mms

```
<<<---
% Author M.Ruckert
 LOC Data_Segment
 GREG @
Decimal BYTE 1,2,3,4,5,6,7,8,9,0,1
 LOC Decimal+32
Binary OCTA 0

 LOC #100
Main IS @
j IS $0
U IS $1
u IS $2
tmp IS $3
m  IS 19

 LDA U,Decimal
 LDO u,Binary
--->>>
 SET j,m-1   $j\is m-1$.
 LDBU u,U,m   $u\is U_m$.
1H MULU u,u,10
 LDBU tmp,U,j; ADDU u,u,tmp $u\is 10u+U_j$.
 SUB j,j,1
 PBNN j,1B   Repeat for $m>j\ge 0$.
```

method1b.plm

```
# Decimal to binary conversion by Horner's method (TAOCP 4.4, Met...
# Input: decimal digits in U[0]..U[m] (most significant in U[m])

# Based on an .mms file by M.Ruckert
segment data;
greg $DataBase;
byte Decimal 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1;   # "12345678901" - 11 digits
segment code;

register $j, $U, $u, $tmp;
const m = 19;# number of digits

$U = $DataBase;
$j = m - 1;
$u = $U[m].bu;   # u = most significant digit (index m)
do:
  $u = $u *u 10;
  $tmp = $U[$j].bu;
  $u = $u +u $tmp;   # u = 10*u + U[j]
  $j = $j - 1;
end while likely $j is nonneg;

trap 0, HALT, 0;
```

method2a: Radix conversion – left-to-right, method 2a

method2a.mms

```
<<<---
% Author M.Ruckert
 LOC Data_Segment
 GREG @
Decimal BYTE 0
 LOC Decimal+32
Binary OCTA 1234567890

 LOC #100
Main IS @
j IS $0
U IS $1
u IS $2
x IS $3
tmp IS $4
n IS 19
ten19 GREG 10*10*10*10*10*10*10*10*10*10*10*10*10*10*10*10*10*10*10
 LDA U,Decimal
 LDO u,Binary
 SUBU u,ten19,1

 CMPU tmp,u,ten19; BNN tmp,Error
--->>>
 PUT rD,u; DIVU x,ten19,ten19 $\.{rR}\is r$ and $x\is\lfloor (wu+10^n)/10^n\rfloor$
 SET j,n-1   $j\is n-1$
OH MULU x,x,10   $x is 10x$
 GET tmp,rH; STB tmp,U,j  $U_j \is \lfloor 10x \rfloor $
 SUB j,j,1   $j \is j-1$
 PBNN j,OB   Repeat for $n>j\ge0$.
<<<---
 SET $255,0; TRAP 0,Halt,0
Error SET $255,1; TRAP 0,Halt,0


--->>>
```

method2a.plm

```
# Binary to decimal (19 digits) via scaled division (TAOCP 4.4, M...
# Uses PUT $rD + DIVU to get the scaled representation, then MULU...
# x = floor((u * 2^64 + $ten19) / $ten19)  - the 19-digit scaled ...

# Based on an .mms file by M.Ruckert
segment data;
greg $ten19 = 10000000000000000000;   # 10^19
greg $DataBase;
byte Decimal 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
octa Binary 1234567890;
segment code;

register $j, $U, $u, $x, $tmp;
$U = $DataBase;
$u = Binary[0].o;

# D1: PUT $rD = u, then DIVU x = ($rD:$ten19) / $ten19 = floor((u...
$rD = $u;
$x = $ten19 /$u $ten19;   # x = floor((u * 2^64 + $ten19) / $ten19)

$j = 18;   # n - 1 = 18
do:
  $x = $x *u 10;   # x = 10x; high digit goes to $rH
  $tmp = $rH;
  $U[$j].b = $tmp;   # U[j] = floor(10x / 2^64) - next most-significant digit
  $j = $j - 1;
end while likely $j is nonneg;

trap 0, HALT, 0;
```

method2aa: Radix conversion – left-to-right, method 2a (accumulated)

method2aa.mms

```
<<<---
% Author M.Ruckert

 LOC Data_Segment
 GREG @
Decimal BYTE 0
 LOC Decimal+32
Binary TETRA 123456789

 LOC #100
Main IS @
j IS $0
U IS $1
u IS $2
x IS $3
tmp IS $4
n IS 9
ten9 GREG 10*10*10*10*10*10*10*10*10
 LDA U,Decimal
 LDTU u,Binary
% SUBU u,ten9,1

 CMPU tmp,u,ten9; BNN tmp,Error
--->>>
 SLU u,u,32; ADD u,u,ten9; DIVU x,u,ten9 $x\is\lfloor (wu+10^n)/10^n\rfloor$
 SET j,n-1   $j\is n-1$
0H 4ADDU x,x,x; SLU x,x,1    $x is 10x$
 SRU tmp,x,32; STB tmp,U,j  $U_j \is \lfloor 10x \rfloor $
 ANDNMH x,#FFFF   $x\is x \bmod w$
 SUB j,j,1   $j \is j-1$
 PBNN j,0B   Repeat for $n>j\ge0$.
<<<---
 SET $255,0; TRAP 0,Halt,0
Error SET $255,1; TRAP 0,Halt,0


--->>>
```

method2aa.plm

```
# Binary to decimal (9 digits, tetra) via scaled division (TAOCP ...
# Uses SLU + ADD to set up 64-bit dividend, DIVU for scaled quoti...
# x = floor((u * 2^32 + $ten9) / $ten9)  using the tetra u.

# Based on an .mms file by M.Ruckert
segment data;
greg $ten9 = 1000000000;   # 10^9
greg $DataBase;
byte Decimal 0, 0, 0, 0, 0, 0, 0, 0, 0;
reserve 22;   # match the original Decimal+32 placement for Binary
tetra Binary 123456789;
segment code;

register $j, $U, $u, $x, $tmp;
$U = $DataBase;
$u = Binary[0].tu;   # load unsigned 32-bit value

goto Error if $u >=u $ten9;

# Compute x = floor((u * 2^32 + $ten9) / $ten9)
$u = $u <<u 32;   # u = u * 2^32
$u = $u + $ten9;   # u = u * 2^32 + $ten9
$x = $u /u $ten9;   # x = floor(u / $ten9) - using DIVU with $rD=0

$j = 8;   # n - 1 = 8
do:
  $x = $x 4addu $x;   # x = 5x
  $x = $x <<u 1;   # x = 10x (multiply by 10 without MUL)
  $tmp = $x >>u 32;   # extract high 32 bits = digit
  $U[$j].b = $tmp;   # store digit
  $x andnmh= #FFFF;   # clear bits 32..47 - keep only low 32 bits in x
  $j = $j - 1;
end while likely $j is nonneg;

trap 0, HALT, 0;

Error:
$255 = 1;
trap 0, HALT, 0;
```

TAOCP 4.5.2 – Greatest Common Divisors

euclid: Euclid's GCD algorithm

euclid.mms

```
<<<---
% Author M.Ruckert
 LOC #100
 GREG @
U OCTA 1000
V OCTA 30

Main LDO $1,U
 LDO $2,V
 PUSHJ $0,gcd
 TRAP 0,Halt,0


--->>>
u IS $0   Parameter
v IS  $1
tmp IS $2   Local variable
Loop DIVU tmp,u,v
 SET u,v   $u\is v$
 GET v,rR   $v\is u\bmod v$
gcd PBNZ u,Loop
 POP 1,0   Done if $u=0$.
```

euclid.plm

```
# Euclidean GCD algorithm using unsigned division (TAOCP 4.5.2, A...

# Based on an .mms file by M.Ruckert
def gcd($u, $v):
  register $tmp;
  while likely $v != 0:
    $tmp = $u /u $v;
    $u = $v;
    $v = $rR;
  end
  return $u;
end

segment data $DataBase;
octa U 1000;
octa V 30;
segment code;
$1 = U[0].o;
$2 = V[0].o;
$0 = call gcd($1, $2);
trap 0, HALT, 0;
```

euclidb: Binary GCD algorithm

euclidb.mms

```
<<<---
% Author M.Ruckert
 LOC #100
 GREG @
U OCTA 1000
V OCTA 30

Main LDO $1,U
 LDO $2,V
 PUSHJ $0,gcd
 TRAP 0,Halt,0


--->>>
u IS $0   Parameter
v IS  $1
k IS $2   Local variable
t IS $3
gcd SET k,0   B1. Find powers of 2.
0H OR t,u,v; PBOD t,2F Both even?
 SR u,u,1; SR v,v,1  $u\is u/2$ and $v\is v/2$
 ADD k,k,1   $k \is k+1$
 JMP 0B
2H NEG t,v   B2. Initialize.
 BOD u,4F
 SET t,u
3H SR t,t,1   B3. Halve $t$.
4H BEV t,3B   B4. Is $t$ even?
 CSP u,t,t   B5. Reset $\max(u,v)$.
 NEG t,t
 CSNN v,t,t
 SUB t,u,v   B6. Subtract.
 PBNZ t,3B
 SL u,u,k
 POP 1,0   Return $u2^k$.
```

euclidb.plm

```
# Binary GCD algorithm (TAOCP 4.5.2, Algorithm B)

# Based on an .mms file by M.Ruckert
def gcd($u, $v):
  register $k,$t;
  $k = 0;
  # B1: Find common factors of 2 - loop while both even
  B1Loop:
  $t = $u | $v;
  goto B2 if likely $t is odd;
  $u = $u >> 1;
  $v = $v >> 1;
  $k = $k + 1;
  goto B1Loop;
  # B2: Initialize t
  B2:
  $t = - $v;
  goto B4 if $u is odd;
  $t = $u;
  # B3: Halve t (signed shift right)
  B3:
  $t = $t >> 1;
  # B4: Is t even? If so, keep halving
  B4:
  goto B3 if $t is even;
  # B5: Reset max(u,v) using conditional sets
  $u = $t if $t is pos;
  $t = - $t;
  $v = $t if $t is nonneg;
  # B6: Subtract; loop until u == v
  $t = $u - $v;
  goto B3 if likely $t is nonzero;
  $u = $u << $k;
  return $u;
end

segment data $DataBase;
octa U 1000;
octa V 30;
segment code;
$1 = U[0].o;
$2 = V[0].o;
$0 = call gcd($1, $2);
trap 0, HALT, 0;
```

TAOCP 4.6.3 – Exponentiation (Exercise 2)

xnmodw: Binary exponentiation (square-and-multiply method)

xnmodw.mms

```
<<<---
% Author M.Ruckert

 LOC #100
Main SET $1,3
 SET $2,4
 PUSHJ $0,Exp
 TRAP 0,Halt,0


--->>>
x IS $0 Parameter
n IS $1
y IS $2 Local variable
z IS $0 Alias for $x$
&
Exp SET y,1 1 A1. Initialize.
 JMP 2F 1
&
0H SRU n,n,1 $L+1-K$ A2. Halve $N$. $n$ even.
5H MULU z,z,z $10L$ A5. Square $Z$.
2H PBEV n,0B $L +1+2K$ A2. Halve $N$.
 SRU n,n,1 K $n$ odd.
 MULU y,z,y $10K$ A3. Multiply $Y$ by $Z$.
 PBNZ n,5B $K+2$ A4. $N=0$?.
 SET $0,y 1
 POP 1,0 3
```

xnmodw.plm

```
# Binary exponentiation: y = x^n mod 2^64 (TAOCP 4.6.3, Exercise ...
# Parameters: x=$0 (base), n=$1
# Returns: result in $0

# Based on an .mms file by M.Ruckert
def Exp($x, $n):
  register $z = $0;# z is same register as x
  register $y = $2;# local accumulator
  $y = 1;
  goto A2;
  # A2: Halve N (n even)
  A2even:
  $n = $n >>u 1;
  # A5: Square Z
  A5:
  $z = $z *u $z;
  # A2: Halve N - check parity
  A2:
  goto A2even if likely $n is even;
  $n = $n >>u 1;
  # A3: Multiply Y by Z
  $y = $z *u $y;
  # A4: N=0?
  goto A5 if likely $n is nonzero;
  return $y;
end

$1 = 3;
$2 = 4;
$0 = call Exp($1, $2);
trap 0, HALT, 0;
```

xnmodws: Sequential exponentiation (repeated multiplication)

xnmodws.mms

```
<<<---
% Author M. Ruckert

 LOC #100
Main SET $1,3
 SET $2,23
 PUSHJ $0,Exp
 TRAP 0,Halt,0


--->>>
x IS $0 Parameter
n IS $1
y IS $2 Local variable

Exp SET y,x 1
 SUB  n,n,1 1
 JMP 1F 1
0H MUL y,y,x $10(n-1)$
 SUB n,n,1 $(n-1)$
1H PBP n,0B $n+2$
 SET $0,y 1
 POP 1,0 3
```

xnmodws.plm

```
# Simple sequential exponentiation: y = x^n mod w (TAOCP 4.6.3, E...
# Parameters: x=$0 (base), n=$1
# Returns: result in $0

# Based on an .mms file by M. Ruckert
def Exp($x, $n):
  register $y;

  $y = $x;
  $n = $n - 1;
  while likely $n is pos:
    $y = $y * $x;
    $n = $n - 1;
  end
  return $y;
end

$1 = 3;
$2 = 23;
$0 = call Exp($1, $2);
trap 0, HALT, 0;
```

References

[1] Wijtze de Boer, "PL/MMIX" (literate program, CWEB source and typeset reference). 2025
http://www.iclassici.nl/plmmix/

[2] Donald E. Knuth, *The Art of Computer Programming, Volume 1, Fascicle 1: MMIX—A RISC Computer for the New Millennium*. Addison-Wesley, 2005.
https://mmix.cs.hm.edu/doc/fasc1.pdf

[3] Donald E. Knuth, "MMIX" (literate program, CWEB source and typeset reference). Stanford University, 1999.
https://mmix.cs.hm.edu/doc/mmix-doc.pdf

[4] Donald E. Knuth, "Literate Programming." *The Computer Journal*, 27(2):97–111, May 1984.

[5] Niklaus Wirth, "PL/360, A Programming Language for the 360 Computers." *Journal of the ACM*, 15(1):37–74, January 1968.

Index

.mmo files, 5
.mms files, 5
.plm files, 5
.plw file format, 44
.plw files, 44
#line directives, 44
$rA (arithmetic status register), 30
$rA register
    exception bits, 31
$rB (bootstrap register), 31
$rBB (bootstrap register), 31
$rC (continuation register), 31
$rD (dividend register), 30
$rE (epsilon register), 30
$rF (failure location register), 30
$rG (global threshold), 30
$rH (himult register), 30
$rI (interval counter), 30
$rJ (return-jump register), 30
$rK (interrupt mask register), 31
$rL (local threshold), 30
$rM (multiplex mask), 30
$rN (serial number), 30
$rO (register stack offset), 31
$rP (prediction register), 31
$rQ (interrupt request), 31
$rR (remainder register), 30
$rS (register stack pointer), 31
$rT (trap address register), 31
$rTT (dynamic trap address), 31
$rU (usage counter), 30
$rV (virtual translation), 31
$rW (where-interrupted), 31
$rWW (where-interrupted trap), 31
$rX (execution register), 31
$rXX (execution register trap), 31
$rY (Y operand), 31
$rYY (Y operand trap), 31
$rZ (Z operand), 31
$rZZ (Z operand trap), 31
16ADDU (MMIX), 24
2ADDU (MMIX), 24
4ADDU (MMIX), 24
8ADDU (MMIX), 24

`Address`, 26
address loading, 28
address-of operator, 28
advanced topics, 6
ANDNH (MMIX), 29
appendix
    bootstrap compiler woven output, 187
    bootstrap tangle woven output, 72
    bootstrap weave woven output, 102
    MMIX side-by-side programs, 393
    TPK woven output, 61
arithmetic operations, 24
Asbjørnsen, Peter Christen (1812–1885), 41
assignment, 24
Augustine, Aurelius Augustinus Hipponensis, Saint (354–430), 38
auto-GREG, 28

base-register selection, 28
`begin`, 16
begin block, 16
Bible
    Book of Genesis, 24
    Proverbs, 72
binary operators, 24
bit manipulation, 29
bitwise operations, 24
branch hints, 35
branching, 33
`break`, 35
buffer declaration, 18
`byte`, 19
byte (data size), 27

cache operations, 38
cache prefetch, 38
`call`, 15
Carroll, Lewis [Charles Lutwidge Dodgson] (1832–1898), 33
Cervantes Saavedra, Miguel de (1547–1616), 393
character literals, 7, 8
`CharLit`, 7
chunk definitions, 44
chunk references, 44, 45
Claude Opus, 5
Claude Sonnet, 5
code chunks, 44
code segment, 10
comments
    line comment, 7
compare-and-swap, 39
compilation workflow, 6
compile-time constants, 12

conditional set (CS), 26
conditions, 35
Confucius [Kong Qiu] (551–479 BCE), 12
const, 12
continue, 35
control flow, 33
cross-reference index, 45
cross-reference links, 45
CS/ZS family, 26
cswap, 39
CSWAP (MMIX), 39
CWEB, 44

data layout directives, 18
data segment, 10
    base address, 20
    procedures in, 20
de Boer, W., 5
def, 14
Descartes, René du Perron (1596–1650), 14
Dijkstra, Edsger Wybe (1930–2002), 5
distribution, 5
do, 34
do...while statement, 34
Don Quixote, 393
DVWIOUZX, 31

elif, 33
Eliot, Thomas Stearns (1888–1965), 58
else, 33
end, 14
entry point, 10
error mapping, 44
exception handling, 41

float conversion, 25
floating-point
    greg literal, 18
    short float, 27
floating-point compare, 35
floating-point operations, 24
Forster, Edward Morgan (1879–1970), 44
Frost, Robert Lee (1874–1963), 22

GET (MMIX), 30
GETA (MMIX instruction), 28
GO (MMIX), 37
goto, 37
goto statement, 37
GPT, 5
grammar, 58
greg, 18
    buffer declaration, 18
    float literal, 18
GREG (MMIX directive), 18
greg+data idiom, 19
greg+reserve idiom, 21

high-tetra (data size), 27

identifiers, 7
if, 33
    postfix, 33
if statement, 33
immediate constants, 7
include, 22
inline string literal, 16
integer compare, 35
IS (MMIX directive), 12

JMP (MMIX), 37

keywords
    reserved, 8
Knuth, Donald Ervin (1938–), 5
Korzybski, Alfred Habdank Skarbek (1879–1950), 43

label definitions, 45
label-based addressing, 28
labels, 7
language syntax, 58
LDUNC (MMIX), 26
ldvts, 43
LDVTS (MMIX), 43
lexical conventions, 7
likely, 35
line comments, 44
literal pool, 20
literate programming, 44
    comments, 44
    TPK example, 45
literate programming tools, 44
load, 26
loc, 20
    two-segment convention, 10
+N (MMIX directive), 20
LOC (MMIX directive), 20
    named segments, 19
local, 22
LOCAL (MMIX directive), 22
local registers, 22
local scope, 16
loop, 34
loop statement, 34

main body

implicit, 10
memory access, 26
memory fence, 38
memory layout, 10
memory ordering, 38
Metamorphoses (Ovid), 187
`mmix`
simulator, 5, 6
MMIX (Knuth's architecture), 5
MMIXAL
Algorithm B, 28
`mmixal`, 5, 6
Moe, Jørgen (1813–1882), 41
multiple return values, 14

named chunks, 44
namespace directives, 22
NNIX simulator, 32

`octa`, 19
octa (data size), 27
ORH (MMIX), 29
ORMH (MMIX), 29
output parameters, 14
Ovid (Publius Ovidius Naso, 43 BC–17/18 AD), 187

page table entry (PTE), 42
parameter registers, 7
PL/360, 5
PL/MMIX, 5
alternative designs, 5
`plmmix-compiler`, 5
plmmix-tangle, 44
`plmmix-tangle`, 44
plmmix-tangle output, 44
plmmix-weave, 44
output, 45
`plmmix-weave`, 44
polarity test, 35
`pool`, 20
postfix if, 33
predefined constants, 30, 31
`prefix`, 22
PREFIX (MMIX directive), 22
`prego`, 38
PREGO (MMIX), 38
`preld`, 38
PRELD (MMIX), 38
`prest`, 38
PREST (MMIX), 38
privileged instruction, 30
procedure calls, 15, 33
procedure declarations, 14
procedure placement, 20
program structure, 10
prose sections, 44
PUSHGO (MMIX), 15
PUSHJ (MMIX), 15
PUT (MMIX), 30

Raymond, Eric Steven (1957–), 18
`register`, 12
register alias transparency, 12
register declarations, 12
auto-register, 12
register usage, 45
registers
general-purpose, 7
`RegOperand`, 24
`reserve`, 20
reserved words, 8
`resume`, 41, 42
RESUME (MMIX), 42
predefined constants, 32
`return`, 15
value expression, 15
return expression, 15
`rounding`, 26
rounding mode, 26
ROUND_CURRENT, 31

`save`, 40
SAVE (MMIX), 40
scaled unsigned add, 24
Schwarzenegger, Arnold Alois (1947–), 40
`segment`, 10, 19
segment base address, 20
Seneca, Lucius Annaeus (c. 4 BC–65 AD), 61
shift operations, 24
short float, 27
`SmallImm`, 7
software distribution, 5
source composition, 22
special registers, 30
Tier 1, 30
Tier 2, 30
Tier 3, 30
Spinden, Herbert Joseph (1879–1967), 102
statements, 24
STCO (MMIX), 26
Stevenson, Robert Louis Balfour (1850–1894), 39
store, 26

string literal
    inline, 16
string literals, 8
    data declarations, 19
structured assembly languages, 5
STUNC (MMIX), 26
sync, 38
SYNC (MMIX), 38
    predefined constants, 32
syncd, 38
SYNCD (MMIX), 38
syncid, 38
SYNCID (MMIX), 38
System/360, 5

tangling, 44
temporary registers, 7
tetra, 19
tetra (data size), 27
Tewa Pueblo
    traditional song, 102
TLB, 43
top-level declarations, 10
top-level statements, 10
TPK algorithm, 45
translation lookaside buffer, 43
trap
    NNIX codes, 32
trap, 41
TRAP (MMIX), 41
traps, 41
trip, 41
TRIP (MMIX), 41
trips, 41

unary operators, 25
Unicode
    character literals, 8
    identifiers, 7
    string literals, 8
unlikely, 35
unsave, 40
UNSAVE (MMIX), 40
UTF-8, 8

virtual memory, 42
virtual translation, 43

weave output, 45
weaving, 44
while, 34
while statement, 34
Wirth, Niklaus Emil (1934–), 10
Wittgenstein, Ludwig Josef Johann (1889–1951), 7
wyde, 19
wyde (data size), 27
wyde-immediate operations, 29
`WydeImm`, 7

zero set (ZS), 26
zero-initialised buffers, 20